UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Rejane Elsing

UMA ABORDAGEM DISTRIBUÍDA PARA UM SISTEMA DE ARQUIVOS NO AMBIENTE CLUX

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Luis Fernando Friedrich

Orientador

Florianópolis, fevereiro de 2005.

UMA ABORDAGEM DISTRIBUÍDA PARA UM SISTEMA DE ARQUIVOS NO AMBIENTE CLUX

Rejane Elsing

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Prof. Raul Sidnei Wazlawick, Dr.

Banca Examinadora

_____

Prof. Luis Fernando Friedrich, Dr. (Presidente)

_____

Prof. Rômulo Silva de Oliveira, Dr.

_____

Prof. José Mazzuco Junior, Dr.

_____

Prof. Frank Augusto Siqueira, Dr.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE SIGLAS

ADSA	Abordagem Distribuída do Sistema de Arquivos
AFS	*Andrew File System*
API	*Application Program Interface*
AVSG	*Availability Volume Storage Group*
DIB	*Directory Information Base*
DIT	*Directory Information Tree*
DN	*Distinct Name*
DNS	*Domain Name System*
DSA	*Directory Service Agent*
DUA	*Directory User Agent*
FLIP	*Fast Local Internet Protocol*
FQDN	*Fully Qualified Domain Name*
GNC	Gerente do Nó de Controle
GNTR	Gerente do Nó de Trabalho Real
HV	Hub Virtual
IP	*Internet Procotol*
IPC	*Interprocess Comunication*
LAN	*Local Area Network*
MAN	*Metropolitan Area Networks*
MIMD	*Multiple Instruction Multiple Data*
MPI	*Message Passing Interface*
NC	Nó de Controle
NFS	*Network File System*
NT	Nó de Trabalho
NTR	Nó de Trabalho Real
NTV	Nó de Trabalho Virtual
PVM	*Parallel Virtual Machine*
RDN	*Relative Distinct Name*
RPC	Chamada a Procedimento Remoto

SA	Sistema de Arquivos
SAC	Servidor de Arquivos Centralizado
SAD	Sistema de Arquivos Distribuído
SCI	*Scalable Coherent Interconnect*
SO	Sistema Operacional
SSI	*Single System Image*
SV	*Switch* virtual
TCP	*Transmission Control Protocol*
TCP/IP	*Transmission Control Protocol / Internet Protocol*
UDP	*User Datagrama Protocol*
VFS	*Virtual File System*
VSG	*Volume Storage Group*
WAN	*Wide Area Network*
XDR	*External Data Representation*

RESUMO

Neste trabalho é realizado o projeto e a implementação da distribuição do sistema de arquivos no ambiente do *cluster* Clux, usando um servidor de arquivos, que inicialmente era centralizado e implementado com processos regulares do Linux. O servidor de arquivos foi re-implementado com *threads*, tornando-se um servidor de arquivos *multithreaded*. Visando distribuir e aumentar a disponibilidade do serviço de arquivos, o serviço foi disponibilizado em dois nós de trabalho do *cluster*, onde cada nó é responsável por gerenciar parte do espaço de nomes compartilhado do *cluster.* Consequentemente, houve a necessidade de projetar e implementar um servidor de diretório *multithreaded*, com o objetivo de identificar qual servidor de arquivos, no ambiente do *cluster*, está disponibilizando o arquivo que o processo cliente deseja manipular. O servidor de diretório identifica a localização física dos arquivos no *cluster,* permitindo que um cliente manipule arquivos sem saber sua localização física.

Palavras-chave: *Cluster* de computadores, Sistemas distribuídos, Sistema de Arquivos Distribuído.

ABSTRACT

In this work it is accomplished the project and the implementation of the distribution of the file system for a cluster Clux, using a file server, that initially was centralized and implemented with regular processes of Linux. The file server was re-implemented with threads, becoming a multithreaded file server. Seeking to distribute and increase the readiness of the file service, the service was availabled in two work nodes of the cluster, where each node is responsible for manage it part of the shared name space of the cluster. Consequently, there was the need to project and to implement a multithreaded directory server, with the objective of identifying which file server, in the environment of the cluster, is availabled the file that the client process wants to manipulate. The directory server identifies the physical location of the files in the cluster, allowing a client to manipulate files without knowing its physical location.

Word-key: Cluster of computers, Distributed Systems, Distributed File System.

INTRODUÇÃO

Este capítulo apresenta as motivações e objetivos deste trabalho, bem como a organização do texto. Na seção 1.1 são apresentadas as motivações, na seção 1.2 são descritos os objetivos e na seção 1.3 é apresentada a organização do texto.

1.1 Motivações

A necessidade de interconexão de diversos computadores localizados em diversos locais diferentes e que executam um trabalho em comum, com necessidade de maior troca de informações, maior velocidade de processamento e maior disponibilidade de informações, provoca o surgimento de sistemas cada vez mais transparentes, escaláveis, confiáveis e com maior desempenho. Essa necessidade motivou o surgimento de pesquisas e projetos que levaram ao aparecimento de novos ambientes capazes de reduzir custos e otimizar a obtenção de resultados.

Uma boa relação entre custo e desempenho é a utilização de *clusters*, que são compostos por microcomputadores comuns, interligados por uma rede de interconexão, que são capazes de executar aplicações paralelas e distribuídas [BOG 02].

No Laboratório de Computação Paralela e Distribuída (LaCPaD) do Curso de Pós-Graduação em Ciência da Computação (CPGCC) da Universidade Federal de Santa Catarina (UFSC), está sendo desenvolvido o projeto de montagem de um *cluster* de computadores derivado do multicomputador Crux [COR 99], que é um ambiente destinado a executar uma rede de processos comunicantes.

Atualmente, o *cluster* conta com um servidor de arquivos centralizado, implementado com processos regulares do Linux, que segue o modelo cliente/servidor e usa um mecanismo de comunicação baseado em *sockets* TCP/IP [PLE 01].

Baseado nesse cenário, motiva-se a transformar o servidor de arquivos centralizado num servidor de arquivos *multithreaded* e disponibilizar esse servidor em mais de um nó de trabalho do *cluster.* Além disso, pretende-se implementar um servidor de diretório para identificar o servidor de arquivos que disponibiliza determinada

informação no *cluster*, permitindo a distribuição do serviço de arquivos e o aumento da disponibilidade do sistema.

1.2 Objetivos

O objetivo principal deste trabalho é o projeto e a implementação de uma abordagem distribuída para um sistema de arquivos no ambiente do *cluster* Clux.

O *cluster* Clux, atualmente, já conta com um servidor de arquivos centralizado, que tem a função de atender requisições de processos clientes que desejam acessar e manipular arquivos. Inicialmente, objetiva-se re-implementar o servidor de arquivos, substituindo os processos regulares do Linux por *threads*, criando um servidor de arquivos *multithreaded*. Em seguida objetiva-se distribuir o serviço de arquivos para aumentar sua disponibilidade, disponibilizando o serviço de arquivos em dois nós de trabalho. Cada um desses nós de trabalho disponibilizará parte dos arquivos locais, para compor o espaço de nomes, que será visto de maneira uniforme por todos os processos clientes do *cluster*. Assim, o espaço de nomes poderá ser compartilhado por todos os processos clientes, sendo chamado de espaço de nomes compartilhado.

Para que os processos clientes possam acessar um arquivo do espaço de nomes compartilhado é necessário implementar algum serviço que realize a localização desse arquivo no *cluster*. Desse modo, o outro objetivo desse trabalho é implementar um servidor de diretório *multithreaded*, para identificar o servidor de arquivos que está disponibilizando o arquivo que o processo cliente deseja acessar. O servidor de diretório será disponibilizado num único nó de trabalho do *cluster*, de maneira dedicada, e deverá permitir que um cliente acesse um arquivo no ambiente distribuído, sem que o cliente tome conhecimento de sua localização física, ou seja, a localização do arquivo é transparente para o cliente.

Portanto, quando um processo cliente executar uma chamada de sistema, essa chama o operador correspondente na interface do sistema, que identifica os argumentos da chamada e extrai do argumento nome de caminho absoluto um prefixo. Em seguida a interface do sistema enviará uma mensagem de requisição ao servidor de diretório, com o prefixo e a identificação do serviço. O servidor de diretório identifica o serviço requisitado e usa o prefixo para localizar o IP do servidor de arquivos que disponibiliza o arquivo, retornando-o à interface do sistema. A interface do sistema recebe o IP e usa-

o para estabelecer uma conexão com o servidor de arquivos apropriado. Em seguida, a interface monta uma mensagem, identificando a chamada de sistema realizada pelo cliente e seus argumentos, e envia-a para servidor de arquivos, que a processa e retorna um resultado. A interface do sistema analisa o resultado e repassa-o para o processo cliente.

1.3 Organização do Texto

O capítulo 2 descreve os sistemas distribuídos, inicialmente fazendo uma introdução e abordando as características procuradas para sistemas distribuídos. Em seguida, apresenta a organização de hardware desses sistemas, classificando-os em multiprocessadores e multicomputadores, dando uma ênfase maior aos multicomputadores heterogêneos e apresentando alguns conceitos relacionados a *clusters*. Também descreve a organização em nível de *software* dos sistemas, apresentando os tipos de sistemas operacionais que existem e a definição do modelo cliente/servidor.

O capítulo 3 apresenta os principais conceitos relacionados aos sistemas de arquivos, os principais requisitos e serviços de um sistema de arquivos distribuído, e exemplos de alguns sistemas de arquivos distribuídos.

No capítulo 4 é realizada a descrição do ambiente operacional do *cluster* Clux, especialmente, do servidor de arquivos centralizado.

O capítulo 5 descreve o projeto e a implementação da abordagem distribuída para o sistema de arquivos no ambiente do *cluster* Clux, apresentando a arquitetura proposta para a distribuição do serviço de arquivos e os detalhes da implementação.

O capítulo 6 apresenta três cenários para exemplificar o funcionamento do sistema, os resultados dos testes realizados para a validação do sistema e as principais características do sistema implementado.

No último capítulo são apresentadas as conclusões e contribuições deste trabalho, e as perspectivas futuras.

2 SISTEMAS DISTRIBUÍDOS

Este capítulo pretende ser uma breve revisão de temas relacionados a sistemas distribuídos, à medida que o presente trabalho tem como objetivo projetar e implementar uma abordagem distribuída para o sistema de arquivos num *cluster* de computadores. Na seção 2.1 é feita uma breve introdução aos sistemas distribuídos e na seção 2.2 são definidas as principais características procuradas em um sistema distribuído. Na seção 2.3 é apresentada a organização de hardware, abordando os multiprocessadores, os multicomputadores, os tipos de redes de interconexão usadas e uma breve descrição de *clusters*. Na seção 2.4 são apresentados os tipos de sistemas operacionais que se destacam para ambientes distribuídos e a definição do modelo cliente/servidor.

2.1 Introdução

No início dos anos oitenta, começaram a surgir novas tecnologias, como o desenvolvimento de microprocessadores com um poder de processamento cada vez maior e a invenção das redes locais de alta velocidade. A aplicação destas duas tecnologias resultou na construção de sistemas de computação poderosos, compostos por um grande número de processadores, ligados através de redes de alta velocidade [TAN 01]. Tais sistemas são denominados sistemas distribuídos.

Em geral, um sistema distribuído é uma coleção de computadores independentes que aparece para seus usuários como um único sistema. As diferenças entre os vários computadores e o modo como eles se comunicam, são escondidos dos usuários [TAN 01].

Assim, os fatores que favoreceram o surgimento de sistemas distribuídos foram [TAN 92]: o uso de microprocessadores que ofereciam uma melhor relação de preço e desempenho do que os *mainframes*; maior interoperabilidade das aplicações que envolvem máquinas separadas fisicamente; maior confiabilidade - se uma máquina falha não prejudica todo o sistema; a possibilidade de crescimento do poder de computação; a

possibilidade de compartilhamento de dados, de dispositivos e de carga de trabalho; e, tentar permitir uma maior comunicação entre as pessoas.

2.2 Características Gerais

As principais características procuradas num sistema, para que ele seja considerado um sistema distribuído são: a capacidade de conectar usuários e recursos facilmente, escondendo o fato que recursos são distribuídos através da rede; ser um sistema aberto, escalável, tolerante a falhas e com controle de concorrência. Essas características são descritas abaixo [TAN 01]:

- **Conectando usuários e recursos:** Os sistemas distribuídos são construídos de uma variedade diferente de redes, sistemas operacionais, hardware de computadores, linguagens de programação e implementações de diferentes desenvolvedores. Uma das características procuradas em um sistema distribuído é conectar facilmente usuários a recursos remotos e compartilhar os recursos com outros usuários de modo controlado, para obter economia, desempenho e realizar a troca de informações [TAN 01].
- **Transparência:** Tentar esconder o fato que processos e recursos estão fisicamente distribuídos através de vários computadores, apresentando para os usuários a idéia de um único sistema. Existem vários tipos de transparência num sistema distribuído como [TAN 01]: transparência de acesso, que esconde a diferença de representação dos dados e o modo como os recursos podem ser acessados; transparência de localização, que esconde do usuário a localização física de um recurso no sistema; transparência de migração, que esconde a idéia que um recurso pode ser movido para outra localização; transparência de relocação, onde um recurso pode ser movido para outra localização enquanto está em uso, sem que o usuário saiba; e transparência de replicação, que esconde a existência de várias cópias de um recurso.
- **Abertura:** Um sistema distribuído aberto deve ter capacidade de interagir com os serviços de outros sistemas abertos, concordando com interfaces bem definidas, suportando portabilidade de aplicações e interoperando com

facilidade. É um sistema independente da heterogeneidade do ambiente, como hardware e software [TAN 01].

- **Escalabilidade:** Quando o sistema permanece efetivo perante um significante aumento no número de recursos e usuários. Para tornar um sistema escalável é necessário pensar em soluções para descentralizar dados, serviços e algoritmos, o que pode provocar o aparecimento de problemas na forma de degradação de desempenho. Existem três técnicas para tentar resolver estes problemas. A primeira tenta esconder o atraso da comunicação entre recursos geograficamente dispersos, evitando esperar por respostas de requisições a servidores remotos, tanto quanto possível usando *caching*. A segunda tenta distribuir as requisições por serviço para outros servidores da rede. E a última tenta replicar os componentes para aumentar a disponibilidade [TAN 01].
- **Tolerância a falhas:** Num sistema distribuído, um processo, um computador, ou uma rede, podem falhar independentemente dos outros. É necessário projetar o sistema de tal forma, que ele possa se recuperar de falhas sem afetar o desempenho geral, continuando a operar de modo aceitável enquanto reparações estão sendo feitas [TAN 01].
- **Concorrência:** Os serviços e aplicações disponibilizam recursos que podem ser compartilhados por clientes em um sistema distribuído. A presença de vários clientes num sistema distribuído é uma fonte de requisições concorrentes para os recursos. Cada recurso deve ser projetado para estar seguro em um ambiente concorrente, utilizando algoritmos de controle de concorrência, que garantem que várias transações possam ser executadas simultaneamente [COU 01].

2.3 Organização de Hardware

Considerando que os sistemas distribuídos são constituídos de vários processadores, existem vários modos diferentes de organizar o hardware de tais sistemas, especialmente pelo modo como eles são conectados e como eles se comunicam [TAN 01]. Pode-se dividir todos os computadores em dois grupos [TAN 01]: aqueles que tem memória compartilhada, normalmente chamados

multiprocessadores, e aqueles que não tem memória compartilhada, chamados de multicomputadores. Cada uma dessas categorias pode ser subdividida considerando a arquitetura e a rede de interconexão. Na seção 2.3.1 são apresentados os multiprocessadores, na seção 2.3.2 são apresentados os multicomputadores e na seção 2.3.3 é feita uma breve descrição de *clusters* de computadores.

2.3.1 Multiprocessadores

Os multiprocessadores são arquiteturas fortemente acopladas, compostas por diversos processadores ligados por um barramento interno de alta velocidade. A principal característica desses, é que os processadores compartilham uma mesma memória global. É através desse compartilhamento que os processos podem se comunicar. Topologias desse tipo de rede são classificadas em: barramento, rede de chaveamento ômega e *crossbar* [TAN 01].

O **barramento** consiste de um certo número de CPUs conectadas a um barramento comum, com um módulo de memória. O problema com este esquema é que, quando a quantidade de CPUs se torna maior ocorre uma sobrecarga sobre o barramento, diminuindo o desempenho. Outro problema é a escalabilidade limitada [TAN 01]. A figura 2.1 apresenta o exemplo de um multiprocessador baseado em barramento.

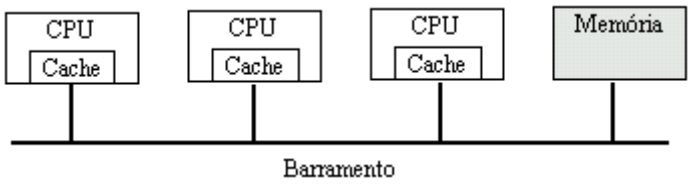


Figura 2.1 – Um multiprocessador baseado em barramento.

Para construir um multiprocessador com uma quantidade maior de processadores, um outro método é usado para conectar a memória aos processadores, chamado *crossbar*. Um ***crossbar*** de tamanho PXM, é formado por P processadores e M módulos de memória, permitindo a conexão simultânea de qualquer processador a qualquer memória, sendo que cada um pode participar de uma conexão por vez [TAN 01]. A figura 2.2 apresenta a estrutura de um *crossbar*.

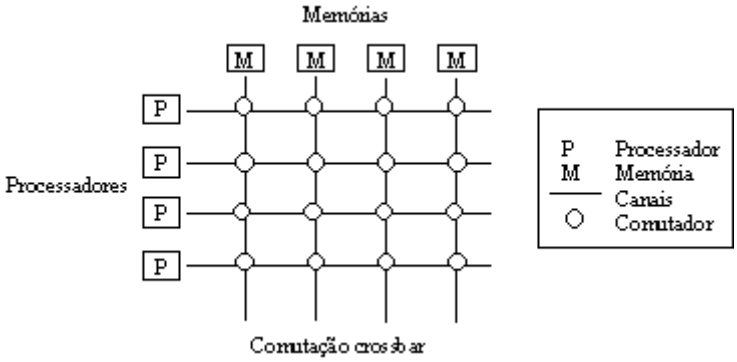


Figura 2.2 – Um multiprocessador baseado em comutação *crossbar*.

As **redes de chaveamento ômega** são constituídas de vários circuitos de chaveamento eletrônico em cada estágio. Os circuitos de chaveamento eletrônico são configurados dinamicamente e permitem o estabelecimento de um caminho direto entre qualquer par processador/memória. Vários estágios aumentam o tempo para conectar um processador a uma memória, o que se torna significativo em redes de muitos estágios [TAN 01]. A figura 2.3 mostra um exemplo de rede de chaveamento ômega com quatro comutadores.

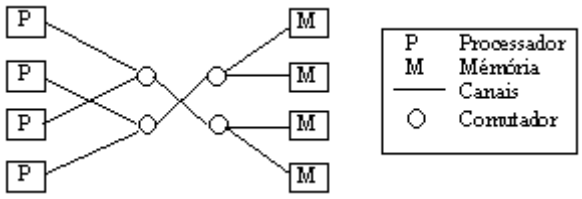


Figura 2.3 – Rede de chaveamento ômega.

2.3.2 Multicomputadores

Um sistema multicomputador é constituído por vários nós, cada um com processador e memória próprios, interconectados por uma rede de comunicação. Multicomputadores são fáceis de construir porque o componente básico consiste em um PC normal com a adição de uma placa de interface de rede, com um ou dois cabos saindo dela. Esses cabos conectam-se a outros nós ou a *switches*, de diversas formas, formando diferentes topologias de interconexão como barramento, estrela, anel, grelha, hipercubo, entre outros [TAN 03].

A interação entre os nós de um multicomputador é alcançada por meio do envio de mensagens que trafegam por redes de interconexão de alta velocidade. As redes de interconexão, que representam o elemento determinante do desempenho do multicomputador, podem ser distinguidas em redes estáticas e redes dinâmicas [COR 99]. Ambas são definidas abaixo:

- **Redes de interconexão estáticas**: Caracterizam-se por comunicar diretamente um determinado nó do sistema com um número de nós predeterminado. O número máximo de conexões de um nó deve ser levado em conta ao se construírem os programas que serão executados pelo multicomputador. A comunicação entre dois nós, que não estejam ligados diretamente, se dá por intermédio de outros nós [BUD 02]. Existem vários tipos de redes estáticas utilizadas em multicomputadores, entre as quais cita-se a grelha e o hipercubo.
- **Redes de interconexão dinâmicas:** Sua principal característica é a possibilidade de alterar suas conexões sem necessidade de intervenção física. Assim, um nó pode utilizar o mesmo meio de comunicação para se comunicar com vários outros nós durante o funcionamento normal dos multicomputadores. Devido a essas características, o roteamento de mensagens raramente é necessário, permitindo estabelecer uma conexão direta entre os nós que desejam se comunicar [BUD 02]. Existem vários tipos de redes dinâmicas, entre as quais pode-se citar o barramento, o *crossbar* e a rede multiestágio, as quais também são topologias usadas nos multiprocessadores.

Os sistemas multicomputadores podem ainda ser subdivididos em multicomputadores *homogêneos* e *heterogêneos* [TAN 01].

2.3.2.1 Multicomputadores Homogêneos

Os *multicomputadores homogêneos* tem somente uma única interconexão de rede, que usa a mesma tecnologia para conectar os nós. Todos os nós são iguais e geralmente tem acesso à mesma quantidade de memória privada. Nos multicomputadores baseados em barramento as mensagens são trocadas entre os nós por

broadcast, e naqueles baseados em comutação, as mensagens entre os nós são trocadas por roteamento [TAN 01].

Como exemplo de multicomputadores homogêneos baseados em comutação tem-se os MPPs (*Massively Parallel Processors*) e os COWs (*Cluster of Workstations*). Os MPPs são supercomputadores compostos de centenas de CPUs, que na maioria dos casos são estações de trabalho ou PCs normais. A diferença desses sistemas de outros multicomputadores é o uso de uma rede de interconexão proprietária de alto desempenho. Os COWs (*Cluster of Workstations*) são uma coleção de PCs padrões ou estações de trabalho, conectados através de um meio de comunicação, tal como Myrinet, que é a interconexão de rede que distingue os COWs dos MPPs [TAN 01].

Duas topologias de rede, comentadas nas bibliografias e que são utilizadas em multicomputadores homogêneos são a grelha e o hipercubo [TAN 01]. A **grelha** é uma das topologias mais conhecidas e utilizadas devido à sua versatilidade. Esta topologia consiste de uma coleção de nós conectados, onde os nós internos têm grau quatro, os nós laterais têm grau três, e os vértices têm grau 2. Uma grelha de grau 4 é mostrada na figura 2.4(a). No **hipercubo,** todos os nós possuem um número uniforme de conexões. Ele possuiu uma dimensão k, ou seja, 2^k nós interconectados de forma que cada nó é ligado a k nós vizinhos. A figura 2.4(b) mostra o formato do hipercubo de dimensão 4.

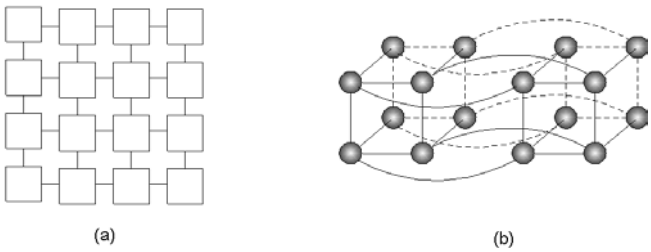


Figura 2.4 - (a) Grelha (b) Hipercubo

2.3.2.2 Multicomputadores Heterogêneos

Os *multicomputadores heterogêneos* podem conter uma variedade de nós independentes e diferentes, que são conectados através de redes diferentes. Isso significa, que os nós conectados ao sistema podem variar o tipo do processador, o tamanho da memória, a largura de banda de I/O e o tipo de interconexão de rede [TAN

01]. Como exemplo, tem-se os *clusters* de computadores, os quais são definidos na seção 2.3.3, e para os quais se aplicam os mesmos princípios definidos na seção 2.3.2.

2.3.3 *Clusters*

Um *cluster* é um conjunto de computadores, interconectados por uma rede de comunicação de alta velocidade, para formar um único computador. Os computadores individuais podem ser computadores pessoais ou estações de trabalho, chamados de nós, com somente um processador ou vários processadores [HOC 03]. Os *clusters* geralmente se diferenciam pelo tipo e número de processadores, o tamanho e velocidade da memória *cache*, o tipo e a velocidade da memória principal, a velocidade do barramento, a qualidade das interfaces de rede, cabeamento, *switches*, entre outros [VAZ 03].

As principais vantagens encontradas no uso de *clusters* são a possibilidade de tornar o sistema mais escalável, tolerante a falhas, com menor custo e independente de fornecedores [VAZ 03].

Os *clusters,* geralmente, são um conjunto de servidores agrupados com intenção de ganho de desempenho, maior disponibilidade e facilidade no gerenciamento. Considerando essas três características, os *clusters* podem ser classificados em três tipos [HOC 03]:

- ***Clusters* de alta disponibilidade:** O *cluster* é projetado com a função de manter a alta disponibilidade do ambiente através de mecanismos de recuperação de falhas, como a duplicação de servidores, ambientes de rede, discos, entre outros. Sistemas de monitoração interna dos *clusters* garantem, que no caso de falha do servidor ativo, o sistema reserva assumirá os serviços automaticamente e instantaneamente.
- ***Clusters* de alto desempenho:** As aplicações devem ser desenvolvidas para execução em ambientes paralelos, que podem envolver um grande número de nós, para a execução de uma grande quantidade de informações dentro de um determinado tempo.
- ***Clusters* altamente escaláveis:** Eles acessam os mesmos sistemas de arquivos, permitem aumento ou diminuição do número de nós de acordo

com a demanda do ambiente e, normalmente, a alta disponibilidade é derivada do balanceamento de carga inteligente [HOC 03].

A representação genérica de um *cluster*, normalmente é composta pelos computadores interligados por *hubs* ou *switches*, como mostrado na figura 2.5.

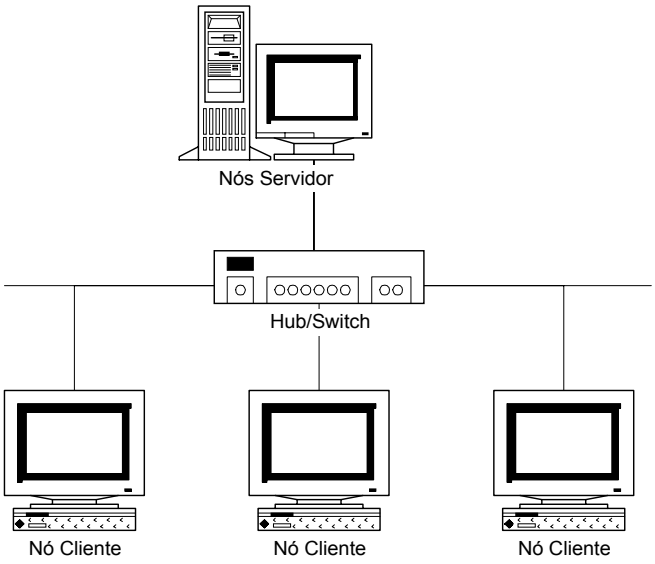


Figura 2.5 - Representação genérica de um *cluster*.

Os c*lusters* podem ser montados rapidamente, utilizando sistemas operacionais de domínio público como o Linux e mecanismos de comunicação de hardware e de software. Entre as tecnologias de hardware, pode-se citar as redes *Ethernet*, SCI e *Myrinet*. Entre as tecnologias de software, os mecanismos de comunicação mais populares são o PVM e o MPI [BUE 02].

2.4 Organização de Software

Os sistemas distribuídos podem ser compostos por máquinas com diferentes tipos de sistemas operacionais, e esses podem ser estruturados de diferentes maneiras. Na seção 2.4.1 são apresentados os tipos de sistemas operacionais que se destacam para ambientes distribuídos, e na seção 2.4.2 é apresentado o modelo cliente/servidor, o qual é visto como um modelo para estruturar os sistemas num ambiente distribuído.

2.4.1 Tipos de Sistemas Operacionais

O sistema operacional de um sistema distribuído compara-se com o sistema operacional tradicional, agindo como gerenciador de recursos para o hardware, permitindo que vários usuários e aplicações compartilhem recursos e tentem esconder a natureza heterogênea e complexa do hardware, provendo uma máquina virtual em que aplicações podem ser facilmente executadas [TAN 01].

Um sistema operacional que roda em um sistema distribuído, pode apresentar as seguintes características [TAN 01]: um maior ou menor grau de transparência; o mesmo sistema operacional ou não, em todos os nós; diferentes unidades de comunicação (pacote, mensagem, arquivo); diferentes números de cópias do sistema operacional; diferentes modos de gerenciamento de recursos; diferente grau de escalabilidade e abertura.

Os sistemas operacionais para sistemas distribuídos podem ser divididos em duas categorias [TAN 01]: sistemas fortemente acoplados e fracamente acoplados. Em sistemas fortemente acoplados, o sistema operacional tenta manter uma visão global única dos recursos gerenciados. Já, em sistemas fracamente acoplados, o sistema operacional pode ser visto como uma coleção de computadores, onde cada um executa seu próprio sistema operacional. Esses sistemas trabalham juntos provendo seus próprios serviços e recursos para os outros.

Considerando as definições feitas acima, pode-se definir basicamente três tipos de sistemas operacionais [TAN 01]: sistemas operacionais distribuídos, sistemas operacionais de rede e sistemas operacionais baseados em *middleware*.

2.4.1.1 Sistemas Operacionais Distribuídos

Os sistemas operacionais distribuídos são sistemas fortemente acoplados, que apresentam basicamente o mesmo sistema operacional em todos os nós. Sua principal meta é ocultar e gerenciar os recursos de hardware de maneira global e distribuída. Eles podem ser divididos em dois grupos [TAN 01]: sistemas operacionais multiprocessadores e sistemas operacionais multicomputadores. Como esse trabalho enfoca multicomputadores serão definidos os sistemas operacionais para multicomputadores.

A estrutura de um sistema operacional distribuído para multicomputadores é apresentada na figura 2.6. Nesse tipo de sistema, o hardware é homogêneo. Cada nó tem seu próprio *kernel* que contém módulos para gerenciar recursos locais tais como memória, CPU, disco, comunicação entre processos, entre outros. A comunicação entre processos geralmente é baseada no envio e na recepção de mensagens entre nós. Acima de cada *kernel* há um nível de software, que implementa o sistema operacional como uma máquina virtual e suporta a execução de várias tarefas de maneira paralela e concorrente [TAN 01].

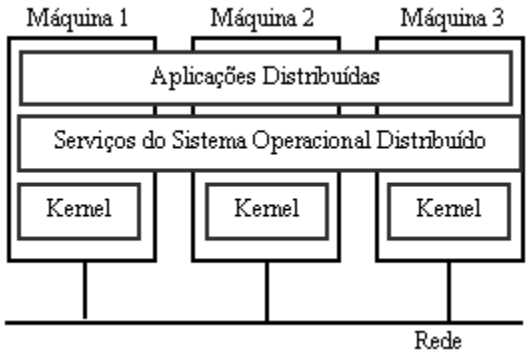


Figura 2.6 – Estrutura de um sistema operacional para multicomputador.

2.4.1.2 Sistemas Operacionais de Rede

Os sistemas operacionais de rede são sistemas fracamente acoplados, usados para sistemas multicomputadores heterogêneos, que podem ser vistos como uma coleção de computadores diferentes, conectados uns aos outros através de uma rede, onde cada um executa seu próprio sistema operacional [TAN 01].

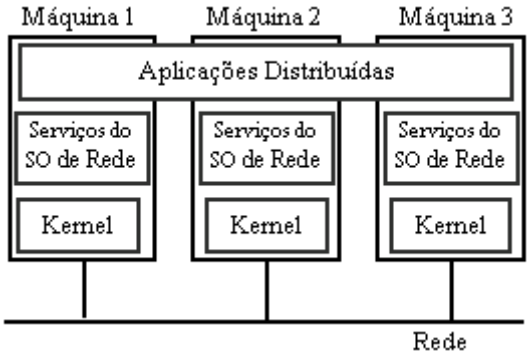


Figura 2.7 – Estrutura de um sistema operacional de rede.

Num sistema operacional de rede, os clientes podem fazer uso de serviços disponíveis em um computador específico. Sua meta é disponibilizar serviços locais para clientes remotos. Geralmente, são sistemas abertos, escaláveis, onde o gerenciamento dos recursos é feito por máquina, a comunicação é realizada através de arquivos, e o grau de transparência é baixo. A estrutura de um sistema operacional de rede é mostrada na figura 2.7 [TAN 01].

2.4.1.3 Sistemas Operacionais baseados em *Middleware*

Um nível adicional sobre a camada do sistema operacional de rede é o *middleware*, como mostra a figura 2.8. Ele implementa serviços de propósito geral, escondendo a heterogeneidade das plataformas subjacentes, provendo transparência de distribuição, atingindo metas de escalabilidade, abertura, fácil relacionamento entre as aplicações e segurança. Exemplos de serviços oferecidos pelo *middleware* são [TAN 01]: serviços de comunicação, serviços que gerenciam informações, serviços de controle e segurança.

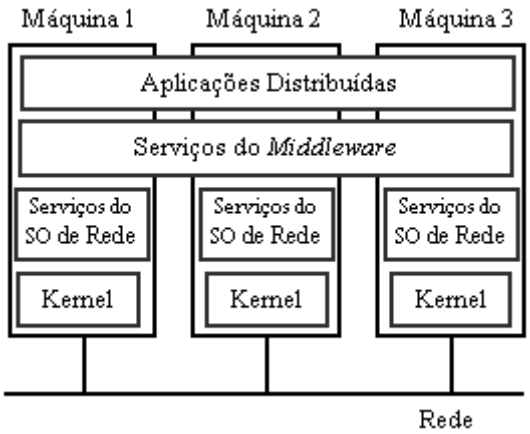


Figura 2.8 – Estrutura geral de um sistema operacional com *middleware*.

2.4.2 Modelo Cliente/Servidor

Entre as diversas maneiras de estruturar um sistema, a mais usada em ambientes distribuídos é o modelo cliente/servidor. Além de servir como modelo para estruturar um sistema, ele também é visto como um padrão de comunicação para sistemas distribuídos.

No modelo cliente/servidor, parte do código do sistema operacional tradicional foi movido para um nível mais alto, deixando o sistema operacional reduzido somente ao *kernel*. As funções tradicionais dos sistemas operacionais como serviço de arquivos, gerência de memória, entre outras, são representadas por processos servidores, que rodam no modo usuário, o que permite um melhor gerenciamento do sistema [TAN 03].

A idéia básica do modelo cliente/servidor para sistemas tradicionais também é usada em sistemas distribuídos, onde os processos são divididos em dois grupos: cliente e servidor. Um processo servidor é um processo implementando um serviço, por exemplo, um serviço de sistema de arquivos. Um processo cliente é um processo que envia uma requisição a um processo servidor e aguarda a resposta desse. A figura 2.9 mostra um exemplo de interação entre um processo cliente e um processo servidor.

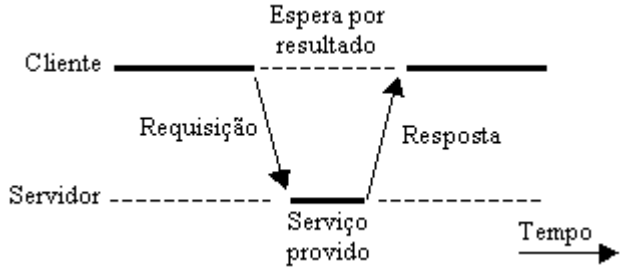


Figura 2.9 – Modelo de interação entre cliente e servidor.

Na figura 2.9, um processo cliente requisita um serviço, empacotando a mensagem e identificando o serviço que deseja. A mensagem é, então, enviada ao servidor. O servidor recebe e desempacota a mensagem, executa a operação associada ao serviço requerido, empacota o resultado e envia a mensagem de resposta ao cliente. A resposta pode conter dados, ou código de erro quando o serviço não pode ser executado.

Num sistema, um nó de trabalho pode rodar: um único processo, vários processos clientes, vários processos servidores, ou uma mistura dos dois. Os processos servidores também podem ser implementados em vários nós de trabalho separados, interagindo o necessário para atender as requisições dos processos clientes. Também pode ocorrer de um processo servidor ser cliente de outro processo servidor [TAN 01]. Por exemplo, um servidor *web* é freqüentemente um cliente de um servidor de arquivos local, que gerencia os arquivos onde as páginas *web* são armazenadas [COU 01].

Esse modelo facilita a manutenção, porque as funções do sistema podem ser isoladas em processos servidores dedicados a serviços específicos, tornando cada parte do sistema menor e mais fácil de implementar [TAN 01].

Num ambiente distribuído a comunicação entre processos clientes e servidores é realizada usando mecanismos como troca de mensagens e chamadas a procedimentos remotos [JUN 96]. No caso de sistemas distribuídos para redes locais, geralmente é adotado o modelo de troca de mensagens, onde consegue-se um bom desempenho, pois a maioria dos níveis dos protocolos em camadas como OSI e TCP/IP são eliminados. Porém, as comunicações remotas apresentam uma complexidade maior de implementação para os desenvolvedores de aplicações, como [TAN 92]: o controle de fluxo de dados, o controle de perda de mensagens e a utilização eficiente de *buffers*. Uma maneira de esconder essa complexidade seria usando o mecanismo de Chamada a Procedimento Remoto (RPC).

3 SISTEMAS DE ARQUIVOS DISTRIBUÍDOS

Este capítulo introduz os principais conceitos relacionados aos sistemas de arquivos distribuídos. Na seção 3.1 é feita uma breve introdução e a apresentação dos principais requisitos de um sistema de arquivos distribuído. Na seção 3.2 são apresentados os principais serviços que compõem um sistema de arquivos distribuído, e na seção 3.3 são descritos alguns sistemas de arquivos distribuídos.

3.1 Introdução

Os sistemas de arquivos foram desenvolvidos originalmente para sistemas centralizados e computadores pessoais, como um componente do sistema operacional, fornecendo uma visão abstrata dos dados armazenados em discos, sendo responsáveis pelo serviço de nomes, acesso a arquivos e de sua organização geral. Com o tempo, eles adquiriram características como controle de acesso e de concorrência, que os tornaram úteis para o compartilhamento de dados e programas [CAR 00].

Um sistema de arquivos distribuído tem o objetivo de fornecer os mesmos serviços e recursos de um sistema de arquivos convencional, pelo menos na visão dos clientes que o acessam, com a diferença que o sistema de arquivos distribuído pode ser acessado de qualquer máquina da rede. Ele suporta o compartilhamento de informações na forma de arquivos e o compartilhamento de recursos de hardware, como armazenamento em disco, através da rede [COU 01].

Um serviço de arquivos bem projetado provê acesso a arquivos armazenados em servidores remotos, com desempenho e confiabilidade similar a arquivos armazenados em discos locais. Atualmente, os serviços de arquivo tentam explorar a maior largura de banda entre redes locais e novos modos de organização em disco, para alcançar maior desempenho, tolerância a falhas, e alta escalabilidade de sistemas de arquivos [COU 01].

Os principais requisitos procurados para um sistema de arquivos distribuído são praticamente os mesmos procurados pelos sistemas distribuídos de modo geral. Assim, os principais requisitos procurados são:

- **Transparência:** Um sistema de arquivos pode ser implementado para suportar diversas formas de transparência como [CAR 00]: transparência de acesso, onde um único conjunto de operações é provido para acessar arquivos remotos e locais; transparência de localização, onde os processos clientes vêem um espaço de nomes uniforme, sem conhecer a localização física dos arquivos; transparência de migração, que esconde a idéia que um arquivo foi movido para outra localização; transparência de desempenho, onde os processos clientes devem continuar a executar satisfatoriamente, enquanto a carga dos servidores varia; e transparência de escalabilidade, onde o serviço pode ser expandido para gerenciar uma área de rede maior e conseqüentemente uma carga maior.
- **Controle de Concorrência:** Vários usuários podem acessar os mesmos arquivos, sem sofrer danos, perdas de desempenho ou quaisquer outras restrições. No caso de haver escrita concorrente é necessário adotar algum modelo de controle para manter a consistência dos dados [COU 01].
- **Replicação de Arquivo:** Um arquivo pode ter várias cópias de seu conteúdo em diferentes localizações. Quando os dados de um servidor são replicados em outro servidor, é possível dividir a carga das requisições clientes entre esses dois servidores, aumentando a disponibilidade dos servidores e a confiança do sistema, e diminuindo o tempo de resposta [COU 01].
- **Heterogeneidade de sistemas operacionais e hardware:** A interface de serviços deve ser definida de tal forma, que os processos clientes e servidores possam ser implementados para diferentes computadores e sistemas operacionais [COU 01].
- **Tolerância a falhas:** É necessário implementar mecanismos de tolerância a falhas de maneira transparente, pois se um processo cliente requisita um arquivo de um servidor que falha, o arquivo não pode ficar indisponível, sendo buscado em outro servidor de *backup* [COU 01].
- **Segurança:** Realizar o controle de acesso ao servidor através da identificação do usuário, e proteger o conteúdo das requisições e mensagens de resposta, usando assinatura digital e criptografia dos dados [COU 01].

3.2 Serviços de um Sistema de Arquivos Distribuído

Os principais componentes de um sistema de arquivos distribuído são o serviço de arquivos, o serviço de diretórios e o serviço de nomes [KON 96]. O serviço de arquivos é responsável por operações relacionadas a arquivos individuais, como leitura, gravação ou remoção de arquivos. O serviço de diretório, por sua vez, está relacionado com operações em diretórios, como a criação e manutenção de diretórios, a adição e remoção de entradas no diretório, entre outras. O serviço de nomes realiza operações para mapear nomes de arquivos, identificar sua localização e seus atributos, entre outras. Abaixo é descrito cada um desses serviços.

3.2.1 Serviço de Nomes

O serviço de nomes cuida de indicar a localização de um determinado arquivo, dado o seu nome ou caminho. Se o nome do arquivo contiver o nome da máquina que possui o arquivo, como por exemplo “jaca:/tmp/teste”, então este serviço de nomes não provê transparência de localização. Para prover essa transparência, o nome de um arquivo não deve ter indícios de sua localização física, e caso esse arquivo mude de lugar, ou tenha várias cópias, o seu nome ou caminho não precisará ser alterado [KON 96].

O serviço de nomes pode oferecer resolução por nomes ou resolução por localização, ou ambos [KON 96]. A resolução por nomes mapeia nomes de arquivos legíveis para humanos, normalmente *strings,* para nomes de arquivos legíveis para computadores, que normalmente são números, facilmente manipuláveis pelas máquinas. A resolução por localização mapeia nomes globais para uma determinada localização. Num sistema distribuído, cada servidor é responsável por resolver a localização de um determinado subconjunto de arquivos. Um exemplo de sistema que realiza um serviço de nomes distribuído é o DNS (Sistema de Nome de Domínio), o qual é descrito abaixo.

Sistema de Nome de Domínio - DNS

O DNS (*Domain Name System*) é um protocolo da camada de aplicação que é parte do padrão TCP/IP. Ele é um esquema de gerenciamento de nomes hierárquico e

distribuído. Define a sintaxe dos nomes usados na Internet, regras para delegação de autoridade na definição de nomes, um banco de dados distribuído que associa nomes a atributos e um algoritmo distribuído para mapear nomes em endereços [SOA 95].

O DNS é um sistema perfeitamente escalável, pois ele é um sistema de banco de dados distribuído que não se degrada à medida que suas bases de dados crescem. Ele propaga automaticamente as novas informações de suas bases de dados para o restante da rede, à medida que essas novas informações são requisitadas em outros domínios [JUN 96].

A estrutura do banco de dados do DNS é semelhante à estrutura de diretórios de um sistema de arquivos, representada por uma árvore invertida. O domínio raiz é implementado por um grupo de servidores de nomes, chamados de servidores raiz. Nenhum dos servidores possui informações completas sobre todos os domínios, mas os seus apontadores podem indicar outros servidores que poderão fornecer a informação desejada [JUN 96]. Um domínio, nada mais é do que uma sub-árvore da estrutura hierárquica de domínios. Cada domínio pode ser dividido em partes menores chamadas subdomínios. Os domínios localizados nas pontas dos ramos da árvore de domínios, geralmente, representam máquinas individuais [FAU 95]. Um nome de domínio completo, que inicia a partir do nome de uma máquina e termina no nó raiz da estrutura hierárquica, é chamado Nome de Domínio Completamente Qualificado (FQDN - *Fully Qualified Domain Name*) [JUN 96]. Por exemplo, suzana.*dsc.ufpb.br.*

Os registros do banco de dados do DNS, que apontam para os servidores de nomes de um domínio, são chamados de registros de servidores de nomes. Esses registros contêm o nome do domínio e o nome da máquina que é servidora do domínio que está sendo apontado [JUN 96]. O administrador da rede é responsável pela atribuição de nomes a endereços numéricos e por armazená-los no banco de dados do DNS [JUN 96]. Por exemplo, o nome suzana.*dsc.ufpb.br* corresponde ao IP 150.165.1.1.

O serviço de nomes do DNS é implementado em cada servidor local e é dividido em dois componentes [JUN 96]: o *resolver* e o servidor de nomes. O *resolver* é uma biblioteca de rotinas de software, que não faz parte do sistema operacional e que é ligada a qualquer aplicação que deseje traduzir endereços. Sua função é receber as requisições clientes e montar requisições de consulta, necessitando para isso conhecer pelo menos o endereço de um servidor de nomes. Ele requisita, do servidor de nomes,

informações a respeito de uma determinada máquina. Na Internet, cada máquina pode ser representada por um nome ou por um endereço numérico único, o endereço IP. O *resolver* traduz esse nome para um endereço numérico IP, ou vice-versa [JUN 96]. Os servidores de nomes contêm parte das informações de um banco de dados, e as tornam disponíveis para os *resolvers*. Se o servidor tiver a resposta para uma requisição, ele responde; caso contrário, ele envia a requisição para outros servidores.

No DNS existem duas situações possíveis: máquinas que são apenas clientes (possuem somente o *resolver*) e máquinas que são clientes e servidores (possuem o *resolver* e o servidor de nomes na mesma máquina) [JUN 96].

Os servidores de nomes podem ser classificados de diferentes maneiras, dependendo de como estejam configurados. Os três principais tipos de servidores de nomes no DNS são [JUN 96]:

- **Servidor primário:** Esse servidor possui informações completas e atualizadas a respeito de seu domínio, e faz a carga dessas informações a partir de um arquivo texto criado pelo administrador de sistemas. Só pode existir um servidor primário para um domínio.
- **Servidor secundário:** É o servidor para o qual a base de dados completa de um domínio é replicada, a partir do servidor primário.
- **Servidor de *caching*:** Recebe as respostas de todas as requisições, que venham de outros servidores de nomes, guardando essas informações para responder, ele mesmo, às consultas que venham a ocorrer.

3.2.2 Serviço de Diretórios

Esse serviço é responsável por manter a organização dos arquivos armazenados no sistema. Ele fornece uma interface para que os usuários possam arranjar seus arquivos num formato hierárquico, que é estruturado em diretórios e subdiretórios [KON 96]. Um diretório pode ter um diretório pai e vários subdiretórios filhos.

Porém, se um diretório permite vários pais, ele não pode ser removido, pois ele pode estar sendo referenciado por outros diretórios ou recursos. Para resolver esse problema, são colocados contadores de referência para arquivos e diretórios. Se o contador chegar a zero, significa que o arquivo ou diretório não tem mais nenhum outro recurso apontando para ele, podendo então ser removido [BAC 86].

Algumas operações oferecidas pelos serviços de diretórios são [KON 96]: criação, remoção, alteração, listagem, entre outras. Além delas, o serviço também influencia no gerenciamento de arquivos, como nas operações de criação, remoção, renomeação, busca, duplicação, entre outras.

Uma questão que deve ser considerada em sistemas de arquivos distribuídos é a visão da hierarquia de diretórios, onde os clientes podem ter, ou não, a mesma visão do sistema de arquivos. Na prática, ambas as técnicas são usadas [JUN 96]. Em sistemas de arquivos distribuídos, como o NFS, o servidor exporta parte do espaço de nomes local para ser compartilhado, e a montagem da árvore de diretórios é feita nos clientes [FRO 94]. Assim, cada cliente tem uma visão diferente do sistema de arquivos.

Um exemplo de serviço de diretório em sistemas distribuídos é o X.500, o qual é apresentado abaixo.

Serviço de Diretório X.500

O serviço de diretório X.500 é um banco de dados, mantido conjuntamente por uma coleção de sistemas abertos e distribuídos, com informações sobre um conjunto de objetos desse sistema. Como exemplos de objetos têm-se arquivos, dados, aplicações, entre outros. Os usuários do diretório, podem ler ou modificar a informação nele armazenada, caso tenham permissão para tal. Sua arquitetura assegura que o diretório pode ser distribuído por uma vasta área geográfica [JUN 96].

Os objetos são armazenados na base de informações de diretório ou DIB (*Directory Information Base*). Essa base é composta por vários registros de entradas, onde cada um possui um conjunto de atributos sobre um determinado objeto. Os registros da DIB são arranjados em uma estrutura de árvore, chamada Árvore de Informação de Diretório ou DIT (*Directory Information Tree*) [JUN 96]. Cada objeto é identificado por um rótulo, denominado nome distinto relativo ou RDN. A lista completa de RDNs no caminho, entre a raiz e a entrada do objeto, é chamado de nome distinto ou DN (*Distinct Name*), o qual identifica de maneira única um registro de entrada no diretório [JUN 96].

O modelo funcional do diretório X.500 é composto pelo [JUN 96]: usuário do diretório, Agente de Usuário de Diretório ou DUA (*Directory User Agent*), Agente de Serviço de Diretório ou DSA (*Directory Service Agent*) e por um ponto de acesso. O

usuário é uma entidade ou pessoa que acessa o diretório. O DUA é apresentado como um processo de aplicação, que representa basicamente um usuário acessando o diretório. Já o DSA é um processo, que é parte do servidor, e que provê zero ou mais pontos de acesso. Um ponto de acesso é onde um serviço é obtido. O diretório pode prover um ou mais pontos de acesso [JUN 96]. Quando o diretório é composto de mais de um DSA, ele é chamado distribuído.

A DIB é subdividida em várias partições, que formam a estrutura da DIT. Um DSA é associado com cada partição, gerenciando e fornecendo acesso a ela [JUN 96]. Quando um usuário deseja acessar o diretório, ele chama o DUA, que resolve a requisição do usuário, comunicando-se com o DSA. Quando o DSA recebe uma requisição de um de seus DUAs locais, ele procura primeiramente em sua partição da DIT, as informações pedidas. Se a informação estiver presente, o DSA responderá diretamente para o DUA; caso contrário, o pedido será passado para outro DSA de nível superior. Um DUA pode interagir com um ou vários DSAs, e os DSAs podem intercomunicar-se com outros DSAs, de modo que possam atender a um pedido. Assim, o diretório sempre retorna o resultado de cada pedido que é feito. Esse resultado pode retornar a resposta esperada ou um erro [JUN 96].

3.2.3 Serviço de Arquivos

O serviço de arquivos é responsável por fornecer operações sobre os arquivos que compõem o sistema. Os arquivos podem ser armazenados de diferentes formas, dependendo do seu tipo e uso. Esse serviço também cuida das propriedades dos arquivos, como data de criação, data de alteração, tamanho, dono do arquivo, permissões de leitura, escrita e execução, além de qualquer outra informação relevante. Ele também mantém a integridade das operações realizadas nos arquivos, por exemplo, quando uma aplicação altera algum arquivo, todas as demais aplicações que estiverem acessando o arquivo devem perceber essa alteração o mais rápido possível [KON 96].

Existem dois tipos de implementação para um serviço de arquivos remoto [KON 96]:

- **Acesso remoto:** O cliente não possui um espaço para guardar os arquivos que estiver usando, e toda e qualquer operação realizada com os arquivos

será sempre através da rede. Esse tipo de acesso necessita de uma rede de interconexão de alta velocidade para impedir que o sistema fique lento.

- **Cópia remota**: O cliente recebe uma cópia do arquivo para trabalhar e quando tiver terminado, devolve as alterações para o servidor. Isso só funciona, se o cliente tiver espaço suficiente para armazenar o arquivo. A velocidade da rede só influenciará durante as transmissões do arquivo de um lado para outro, e a implementação é muito eficiente caso o arquivo seja totalmente alterado. Porém, se o cliente só se interessar por uma determinada porção do arquivo, somente os blocos que ele quer trabalhar são enviados para ele. Isso é chamado de *cache* de bloco.

Também é possível que o serviço de arquivo possua funções adicionais que permitam a replicação de arquivos. Quando arquivos são replicados no mesmo servidor ou num servidor remoto, sua disponibilidade é aumentada, o que ajuda a melhorar o acesso concorrente aos arquivos, aumenta a velocidade das aplicações clientes e a disponibilidade dos arquivos caso algum servidor caia ou fique fora do ar [KON 96].

Na próxima seção são apresentados alguns sistemas de arquivos distribuídos, onde se tem a possibilidade de avaliar suas principais características.

3.3 Exemplos de Sistemas de Arquivos Distribuídos

Para atender ao objetivo do trabalho que é projetar e implementar uma abordagem distribuída do sistema de arquivos para um ambiente de *cluster*, realizou-se um estudo de caso de alguns sistemas de arquivos distribuídos como o Sistema de Arquivos de Rede da *Sun* (NFS), o Sistema de Arquivos Andrew (AFS), o Amoeba e o Sprite, os quais são descritos nas próximas seções. Na seção 3.3.5 é feita uma comparação entre os sistemas estudados, analisando suas principais características.

3.3.1 NFS - Sistema de Arquivos de Rede

O *Network File System* é um sistema de arquivos distribuído, desenvolvido pela *Sun Microsystems,* utilizado por um grande número de usuários e que oferece um

sistema de acesso transparente a arquivos remotos. Foi projetado para redes locais com estações de trabalho baseadas em UNIX, mas também pode ser usado em redes de maior porte [KON 96]. É um sistema heterogêneo, pois foi implementado para a maioria dos sistemas operacionais conhecidos e plataformas de hardware [JUN 96].

O NFS utiliza a interface RPC para realizar a comunicação, porque ela usa uma representação de dados independentes de máquina chamada *External Data Representation*, ou simplesmente XDR. O XDR é um protocolo da camada de apresentação, voltado para suportar a troca estruturada de informações entre computadores que se caracterizam por possuírem plataformas de hardware e software completamente heterogêneas, como é o caso do NFS [JUN 96].

A idéia básica do NFS é permitir que um conjunto de clientes e servidores possam compartilhar um sistema de arquivos comum. Ele permite que um nó seja ao mesmo tempo um cliente acessando arquivos remotos, ou um servidor exportando alguns de seus arquivos [TAN 92].

Arquitetura

A estrutura do NFS se subdivide em níveis, como é apresentado na figura 3.1 [TAN 01]. O primeiro é o nível da interface de chamadas de sistema, que trata das chamadas de sistema, verificando a sintaxe da chamada, validando seus parâmetros e chamando o nível do sistema de arquivo virtual (VFS). O sistema de arquivos virtual (VFS) é um padrão para a interconexão entre diferentes sistemas de arquivos. Ele admite múltiplas implementações de sistemas de arquivos locais, de modo que eles possam ser usados para suportar uma grande variedade de sistemas de arquivos locais e distribuídos. Cada operação do VFS é implementada como uma chamada a procedimento remoto [TAN 01].

O VFS mantém uma tabela para cada sistema de arquivos montado, com uma entrada para cada um dos arquivos abertos chamada nó-v. O nó-v identifica unicamente um arquivo de qualquer ponto da rede. Os nós-v são usados para informar se o arquivo é local ou remoto. Se o arquivo é local, o nó-v contém uma referência para o nó-i do arquivo local. Se o arquivo é remoto, ele contém o manipulador do arquivo que fornece todas as informações necessárias para acessar o arquivo [TAN 01].

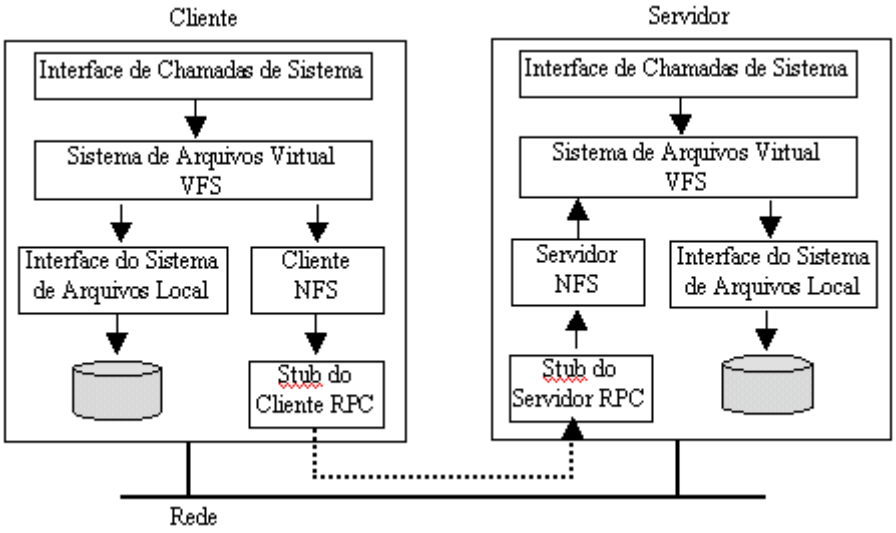


Figura 3.1 – Arquitetura básica do NFS para sistemas UNIX. Fonte: Tanenbaum, A. S., Steen, M. van. *Distributed Systems. Principles and Paradigms*, Prentice Hall, 2001.

Quando uma aplicação faz uma requisição ao serviço de arquivos, esta solicitação é encaminhada para a interface do sistema de arquivos virtual, que de acordo com a localização física do arquivo, remete a requisição para o sistema de arquivos local ou para o cliente NFS. Se o arquivo for local, usa as chamadas de sistema providas pela interface do sistema de arquivos local. Se o arquivo for remoto, acessa-o de forma transparente através do cliente NFS [KON 96]. O cliente NFS implementa as operações do sistema de arquivos, de modo que elas façam a manipulação de acesso a arquivos armazenados em um servidor NFS remoto, usando a interface de comunicação RPC. O servidor NFS processa o pedido, através da execução de chamadas de sistema, usando o sistema de arquivos local, e retorna ao cliente NFS um código de estado e os dados associados com a operação [JUN 96].

Espaço de nomes

O espaço de nomes compartilhado é criado quando um servidor NFS exporta um ou mais de seus diretórios, permitindo que clientes remotos tenham acesso a esses diretórios. Quando um diretório fica disponível, ficam disponíveis também todos os seus subdiretórios. Os clientes NFS geralmente possuem uma árvore de diretórios com raiz privada, na qual eles montam os diretórios do espaço de nomes compartilhado que

são de seu interesse [COU 01]. Como resultado, é possível que clientes tenham acesso aos mesmos arquivos, porém, usando nomes de caminho completamente diferentes [JUN 96].

Quando um nó cliente é inicializado, o seu sistema de arquivos é formado unicamente pelo sistema de arquivos local. Para automatizar o processo de configuração do espaço de nomes, existe o arquivo */etc/fstab*, que contém uma lista de diretórios que são montados automaticamente no momento da inicialização da máquina. Usando o mecanismo de *mount* do UNIX, um cliente pode anexar sub-árvores remotas, exportadas pelos servidores NFS, à sua árvore privada. Para isso, o cliente envia, via RPC, um nome de caminho para um servidor e solicita permissão para montar o diretório especificado em algum lugar na sua hierarquia de diretórios. Se o nome do caminho for legal e o diretório especificado nele tiver sido exportado pelo servidor, ele envia de volta ao cliente uma mensagem com um *manipulador de arquivo* e monta através do uso do *mount* o diretório na estrutura de diretório do cliente. Este *manipulador de arquivo* contém informações como o tipo de sistema de arquivo, disco onde ele se encontra, número do nó-i do diretório e informações sobre segurança. Chamadas subseqüentes de leitura ou escrita no diretório montado são feitas utilizando este *manipulador de arquivo*. Assim, cada cliente verá um espaço de nomes diferente. Um espaço de nomes uniforme pode ser estabelecido com a configuração adequada de tabelas em cada cliente [TAN 01].

Um cliente sem disco pode montar um sistema de arquivo remoto localmente, resultando num sistema de arquivo totalmente suportado por um servidor NFS remoto. 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 [KON 96].

A figura 3.2 apresenta a montagem de uma estrutura de diretório num nó cliente usando o NFS. O servidor B exporta o diretório *Duda*, o qual é montado na hierarquia de diretórios do servidor A. O servidor A exporta o seu diretório *user1,* que contém o subdiretório *Duda*, que é remoto. O cliente monta, na sua estrutura hierárquica local, o diretório remoto *user1* que foi exportado pelo servidor A e que contém o subdiretório remoto *Duda*, exportado pelo servidor B [TAN 01].

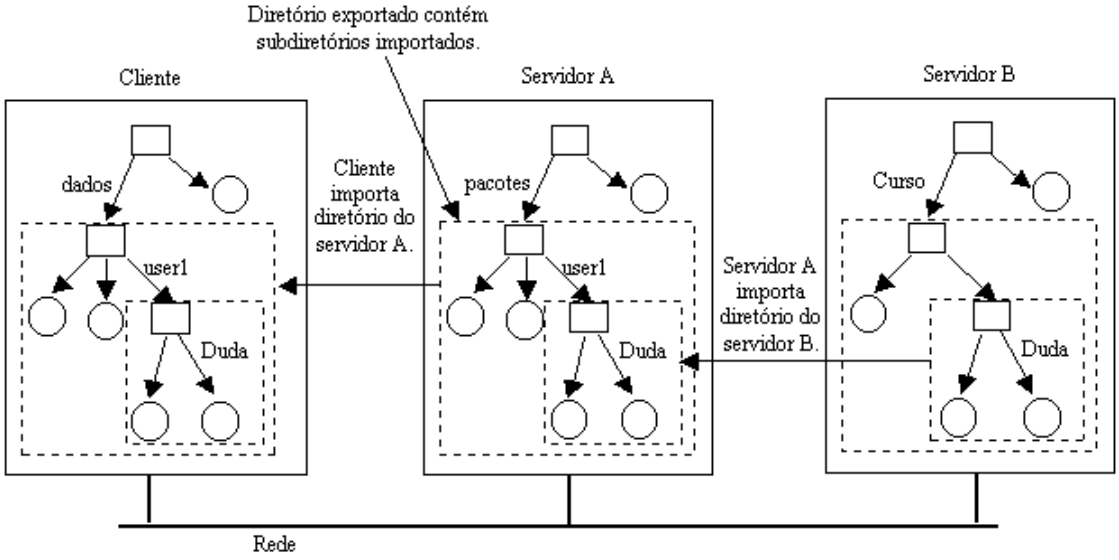


Figura 3.2 – Montando uma estrutura de diretório no NFS.

Servidor sem estado

O NFS foi projetado de tal modo que ele desconsidera o estado dos clientes, ou seja, o servidor NFS é sem estado. A principal vantagem de um servidor sem estado é a simplicidade de implementação de rotinas de recuperação, permitindo que clientes e servidores retomem a execução depois de uma falha, sem a necessidade de qualquer procedimento de recuperação. Além disso, não gasta memória para guardar informações sobre os clientes, podendo assim, oferecer o serviço de arquivos para um número maior de clientes. Assim, cada RPC realizada por um cliente, contém todas as informações necessárias a execução do serviço [COU 01].

Porém, um servidor sem estado não assegura a consistência do sistema de arquivos, porque os seus servidores não controlam os acessos concorrentes para seus arquivos [KON 96]. Isso também torna difícil a gerência de memória *cache*, pois os servidores não tomam conhecimento das operações de *cache* que possam estar ocorrendo em clientes. Clientes diferentes podem ter cópias diferentes e conflitantes do mesmo arquivo ou diretório na sua *cache* local [JUN 96].

Os clientes NFS são geralmente implementados de forma a suportarem pacientemente a falta de resposta de um servidor. Se um servidor cair, o cliente ficará tentando emitir o pedido novamente, até que o servidor responda [JUN 96].

Caching

A implementação de *caching* é indispensável no cliente e no servidor, para alcançar maior desempenho. A *cache* pode conter páginas de arquivos, diretórios e atributos de arquivos. Os clientes são responsáveis por verificar se os dados da *cache* estão atualizados ou não. Na *cache* do cliente é usado um vetor de *timestamp* para validar blocos de *cache*, antes deles serem usados [TAN 01].

Por exemplo, se um cliente guarda um arquivo na memória *cache*, ele deve verificar o estado desse arquivo na *cache* do servidor, antes de cada acesso, para assegurar que o arquivo está atualizado [JUN 96].

Replicação

A especificação original do NFS não suporta replicação de arquivos, porém, versões mais recentes permitem um certo nível de replicação de dados, somente para leitura, através do mecanismo de *automounter*. O *automounter* mantém uma tabela de pontos de montagem, com uma referência para um ou mais servidores NFS. A primeira vez que um arquivo remoto for aberto, o sistema operacional envia uma mensagem a cada um dos servidores NFS. O primeiro que responder vai ter seu diretório montado no cliente [COU 01].

Segurança

Os servidores NFS administram a segurança do sistema através de dois mecanismos. O arquivo */etc/exports* indica quais clientes podem ter acesso e qual tipo de acesso a cada um dos diretórios. Os bits de proteção do UNIX indicam quais usuários e grupos de usuários podem acessar cada arquivo [KON 96].

Versões mais recentes do NFS podem ser configuradas para prover um nível mais alto de segurança, através de um mecanismo de autenticação baseado em chave pública, para processos clientes e servidores, criptografando a informação antes dela ser enviada através da rede [KON 96].

3.3.2 AFS - Sistema de Arquivos Andrew

O projeto do Sistema de Arquivos Andrew (*Andrew File System* - AFS) iniciou na Universidade de *Carnegie Mellon*, em 1983, com suporte IBM. Sua meta foi projetar e implementar um sistema de arquivos distribuído ideal para o ambiente acadêmico, que iria permitir o compartilhamento de uma estrutura de diretório, comum entre centenas de máquinas clientes, de maneira transparente [COU 01].

Arquitetura

Em termos de arquitetura, o AFS possui alguns pontos em comum com o NFS: os clientes usam a interface VFS para ter acesso aos arquivos e o mecanismo de comunicação utilizado é RPCs, o qual utiliza o formato XDR para a representação externa de dados [JUN 96].

Outros componentes da arquitetura do AFS são o *vice* e *venus*. *Vice* é o nome dado ao processo servidor que atende às solicitações de serviço de arquivos que chegam. Desde o AFS-2, o *vice* é *multithreaded*, conseguindo atender várias solicitações simultâneas eficientemente. *Venus* é o nome dado ao cliente AFS, o qual é responsável pela interface entre o cliente e o *vice*. Os nós clientes executam todo tipo de aplicação que os usuários podem precisar. Já os nós servidores executam apenas o *vice* [KON 96].

A figura 3.3 apresenta um *cluster* de computadores que implementa o AFS. O *cluster* é composto por nós servidores dedicados e estações de trabalho clientes. Nos nós servidores o processo servidor em nível de usuário é representado pelo *vice,* e nas estações de trabalho clientes tem-se o *venus* em nível de *kernel.*

Inicialmente implementado como um processo em nível de usuário, o *venus* foi integrado ao *kernel* dos clientes, a fim de melhorar o seu desempenho [KON 96]. Chamadas de sistema como *open, close* e algumas outras, são interceptadas quando elas referem a arquivos do espaço de nomes compartilhado, e são passadas para o processo *venus*. O processo *venus* implementa a estrutura de diretórios hierárquica requerida pelos processos dos usuários UNIX, traduz nomes de caminhos emitidos por clientes para identificadores únicos de arquivos, e gerencia a memória *cache* [COU 01].

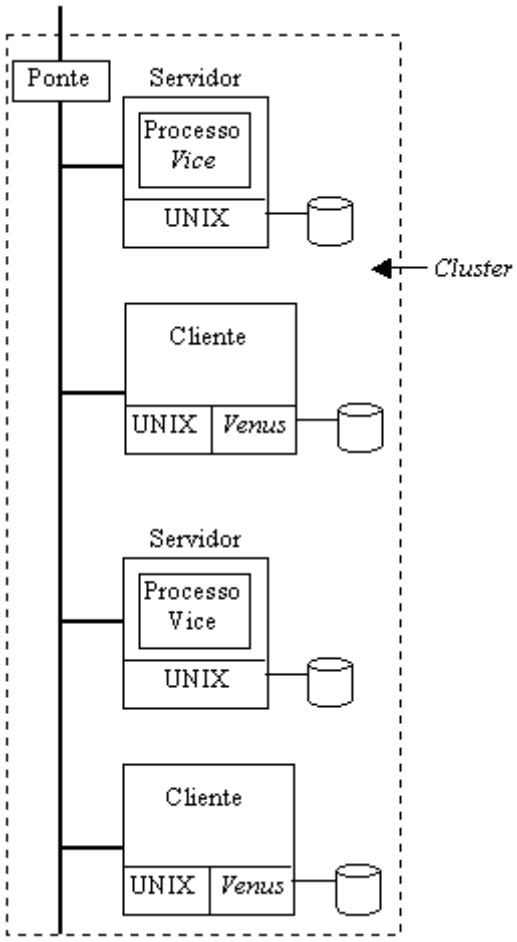


Figura 3.3 – Arquitetura de *cluster* usando o AFS.

Espaço de Nomes

No AFS o sistema de arquivos é apresentado como um espaço de nomes uniforme para todo o sistema, independente de localidade e com hierarquia similar ao UNIX. O espaço de nomes é dividido em duas partes, como mostra a figura 3.4: local e compartilhado. O espaço local é particular a cada nó, e o espaço compartilhado é independente de localidade e comum a todos os nós. Arquivos compartilhados são armazenados em servidores, e cópias deles são armazenadas nos discos locais das estações de trabalho, que contém a aplicação do usuário [COU 01].

O espaço de nomes compartilhado é dividido em sub-árvores, sendo que cada uma delas está inteiramente contida em um único servidor. A uma sub-árvore dá-se o nome de volume. A localização é feita a partir de uma base de dados replicada em todos os servidores, que mapeia nomes de volumes em servidores [COU 01].

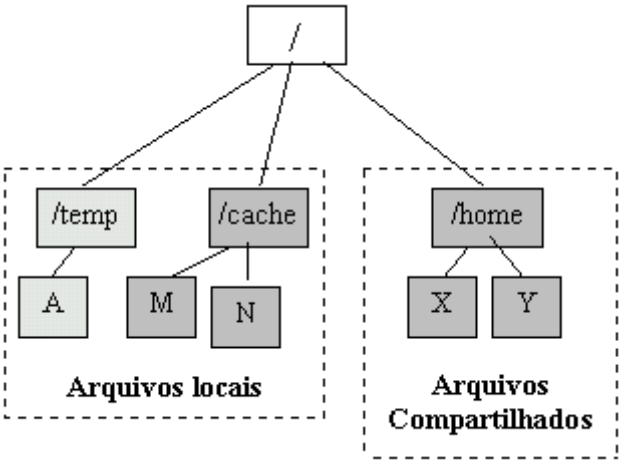


Figura 3.4 – O espaço de nomes visto por um cliente.

Na figura 3.4, o espaço de nomes é dividido em duas partes: arquivos locais e arquivos compartilhados. Os arquivos locais são mantidos nas sub-árvores */temp* e */cache,* armazenadas em discos locais, onde são guardadas informações temporárias e os arquivos necessários para a inicialização da máquina. Os arquivos compartilhados são mantidos por servidores dedicados na sub-árvore /*home* e são iguais para todas as máquinas da rede [KON 96].

Os arquivos são identificados por um identificador de arquivo, que não é visível para os processos clientes e o qual é usado para realizar a comunicação entre clientes e servidores. Quando um processo cliente executa a chamada de sistema *open*, o *venus* verifica se o nome de caminho do arquivo é local ou se está na área compartilhada. Se for local, o sistema de arquivos UNIX local completa o serviço. Se o arquivo está na área compartilhada, o *venus* executa operações de localização do arquivo, resolvendo o nome do caminho componente a componente, percorrendo vários servidores, até encontrar o identificador do arquivo. Em posse do identificador, ele verifica se existe uma cópia válida do arquivo no seu *cache* e toma as providências necessárias [KON 96].

Caching

Por razões de desempenho e escalabilidade, uma das partições no disco local de cada estação de trabalho cliente é usada como uma *cache*, mantendo armazenadas as cópias dos arquivos do espaço de nomes compartilhado [COU 01]. Quando um processo

cliente abre um arquivo, o *venus* tenta descobrir o seu identificador. De posse desse, ele verifica se existe uma cópia válida deste arquivo em sua *cache*. Se não existe, o arquivo inteiro é trazido para a *cache* local. O *venus* retorna, ao processo que abriu o arquivo, um identificador da cópia do arquivo no disco local. A partir daí, todas as operações com este arquivo são realizadas na cópia local. Se o arquivo local estiver desatualizado, uma cópia atualizada é obtida por demanda junto ao servidor correspondente. Somente quando o arquivo é fechado, e se este foi modificado, ele é transferido de volta ao servidor. O servidor atualiza o conteúdo do arquivo e o *timestamp* do arquivo [COU 01]. Assim, os dados alterados apenas se tornam visíveis para outros clientes após o fechamento do arquivo, caracterizando a semântica de sessão [KON 96].

Quando um processo *venus* copia um arquivo remoto para o disco local, o servidor *vice* 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 [COU 01].

A consistência da semântica de sessão entre os clientes é mantida através de um mecanismo chamado *callback*. Quando o *vice* fornece um arquivo para o *venus*, ele também fornece um *callback*, que é uma promessa de que o arquivo, na *cache* do cliente, é a versão mais recente. Os c*allbacks* podem ser quebrados por um cliente, quando ele atualiza a sua cópia local, ou por um servidor, quando ele recebe uma nova versão do arquivo de algum cliente e envia uma mensagem a todos os clientes quebrando o *callback* desse arquivo. A cópia local de um arquivo pode ser usada várias vezes, enquanto o cliente possuir um *callback* válido [KON 96].

Replicação

O AFS suporta um esquema simples de replicação, que permite que volumes sejam replicados apenas para leitura, em vários servidores. Atualizações são feitas somente no volume original e a propagação das réplicas é feita por um procedimento administrativo não automático [JUN 96].

Um cliente AFS sempre acessa a cópia do volume que está mais próxima. Se um servidor não responde as chamadas de um cliente, devido a problemas com a rede ou com ele mesmo, o cliente passa automaticamente a acessar os mesmos dados de uma cópia em outro servidor [KON 96].

Segurança

Para prover acesso seguro aos servidores, o AFS-3 utiliza mecanismos de autenticação e transmissão segura baseada em criptografia. Ele usa o protocolo Kerberos para autenticação mútua entre clientes e servidores. Quando um usuário abre uma sessão, a sua senha é utilizada para estabelecer um canal seguro de comunicação com o servidor Kerberos. Em seguida, o cliente obtém um par de chaves de autenticação, que será usado, no futuro, para estabelecer conexões seguras entre ele e o servidor [COU 01].

O controle de acesso a arquivos é feito através de uma extensão do esquema de bits de proteção do UNIX. Através do uso de uma lista de controle de acesso, associada a cada diretório, é possível identificar quem tem acesso a cada diretório e qual tipo de acesso é permitido [JUN 96].

3.3.3 AMOEBA

O sistema operacional distribuído Amoeba se originou na *Vrije Universiteit de Amsterdam*, em 1981, como um projeto de pesquisa na área de computação paralela e distribuída. Diferentemente da maioria dos sistemas operacionais que começaram baseados em um sistema operacional já existente, o Amoeba começou do nada, desenvolvendo um sistema operacional a partir do zero [FRO 94].

O principal objetivo do projeto foi a construção de um sistema operacional distribuído, totalmente transparente. Todos os serviços do Amoeba estão distribuídos em diversos nós espalhados ao longo da rede, incluindo servidores de processos, servidores de arquivo, servidores de diretórios, servidores de replicação, servidores de *boot*, entre outros. Ele foi projetado para operar com várias arquiteturas e com sistemas heterogêneos [TAN 92].

Arquitetura

O Amoeba é um sistema operacional baseado em objetos e no modelo cliente/servidor. Um objeto é uma estrutura de dados encapsulada, sobre a qual usuários autorizados podem executar funções bem definidas, independentemente da localização

dos usuários e dos objetos. Cada objeto está associado a um servidor que o gerencia [FRO 94].

Os objetos de um ambiente Amoeba são identificados e protegidos por capacidades. Uma capacidade possui quatro campos [TAN 92]:

- **Número da caixa postal do servidor:** Identifica um canal de comunicação entre os clientes e o servidor que implementa os métodos do objeto;
- **Número do objeto:** Usado pelo servidor para identificar o objeto. No caso do servidor de arquivos, pode ser o número do nó-i 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.

Quando um cliente solicita a criação de um objeto, o servidor que gerenciará tal objeto, devolve ao cliente uma capacidade desse objeto, protegida por criptografia. Essa capacidade deve ser apresentada ao servidor, sempre que o cliente desejar executar alguma operação sobre o objeto [FRO 94]. Todos os objetos do sistema, sejam de hardware ou de software, são identificados, protegidos e gerenciados por capacidades [TAN 92].

Na localização de um objeto, se um nó contém um objeto, então ele também contém o servidor que implementa os métodos de manipulação daquele objeto. Dessa 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 [FRO 94].

Para executar uma operação sobre um objeto remoto ou local, os processos clientes utilizam um mecanismo de RPC, implementado no núcleo do sistema operacional [FRO 94]. Outro modelo de comunicação é a comunicação em grupo, que permite que uma mensagem seja enviada, de maneira confiável, de 1 remetente para n receptores. Esse modelo de comunicação pode ser útil para aplicações que replicam dados, a fim de obter tolerância a falhas e manter a consistência dos dados que estão replicados [TAN 92].

O Amoeba usa o protocolo FLIP (*Fast Local Internet Protocol*) para transmitir as mensagens e determinar a localização de processos. Este protocolo apresenta as seguintes características: trata tanto da chamada a procedimento remoto, quanto da

comunicação em grupo; provê suporte a migração de processos; provê suporte de segurança; adapta-se automaticamente a novas configurações de rede; e, funciona em redes de longa distância [TAN 92].

Sistema de Arquivos

O Amoeba considera arquivos como sendo objetos. O sistema de arquivos é implementado por três servidores [TAN 92]: servidor de diretório, servidor de arquivos e servidor de replicação, que são descritos abaixo.

Servidor de Arquivos (Bullet)

O servidor de arquivos também é conhecido como *Bullet* e tem esse nome por ter sido projetado para ser particularmente rápido. Para apresentar um bom desempenho, o *Bullet* trabalha com arquivos imutáveis, ou seja, nenhum arquivo que tenha sido criado pode ser modificado em operações subseqüentes, sendo seu tamanho conhecido no momento de sua criação. Isto facilita várias operações, inclusive a replicação. Permite também, que se aloque os arquivos de forma contígua nos discos, que se faça *cache* de arquivos inteiros em memória principal e que eles sejam transferidos em uma única RPC, reduzindo a fragmentação interna e aumentando o desempenho do servidor [TAN 92].

Como os arquivos 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 esse lhe envie o arquivo na íntegra. O cliente então, modifica o arquivo e o reenvia ao servidor. Essa operação cria um novo arquivo, com uma nova capacidade. Após receber a nova capacidade, o cliente deve solicitar ao servidor que remova o arquivo antigo [TAN 92].

Cada disco sob o controle do *Bullet* possui uma tabela de arquivos, similar à tabela de nós-i do UNIX, com uma entrada para cada arquivo. Essa tabela é carregada na memória quando o servidor é inicializado e lá permanece até a sua desativação. A tabela de arquivos também é utilizada para controle de *cache*. Uma vez que o *Bullet* faz

cache de arquivos inteiros na memória, cada entrada da tabela de arquivos possui um ponteiro para a cópia do arquivo na *cache*, se existente [TAN 92].

Servidor de Diretório

A principal função do servidor de diretório é mapear nomes de objetos em capacidades. Cada diretório é organizado como uma tabela, onde cada linha descreve um objeto. Os diretórios também são objetos e possuem capacidades associadas. As capacidades dos diretórios podem ser armazenadas em outros diretórios, permitindo a formação de árvores hierárquicas [TAN 92].

Ao contrário dos arquivos, os diretórios não são imutáveis. O servidor de diretório provê métodos para criação e remoção de diretórios, inserção e remoção de linhas em diretórios já existentes e pesquisa de nomes em diretórios [TAN 92].

Através do servidor de diretório é possível acessar objetos replicados de forma eficiente. Cada linha de um diretório pode conter várias capacidades, que se referem às réplicas do objeto que são gerenciadas por servidores distintos [FRO 94].

A organização típica de um diretório no Amoeba envolve uma árvore com raiz privada para cada usuário e algumas sub-árvores compartilhadas com os demais usuários. Assim sendo, cada usuários enxerga um espaço de nomes próprio. Por convenção, todas as árvores incluem o diretório *public,* que é o começo do espaço compartilhado de objetos [TAN 92]. A figura 3.5 apresenta esta idéia.

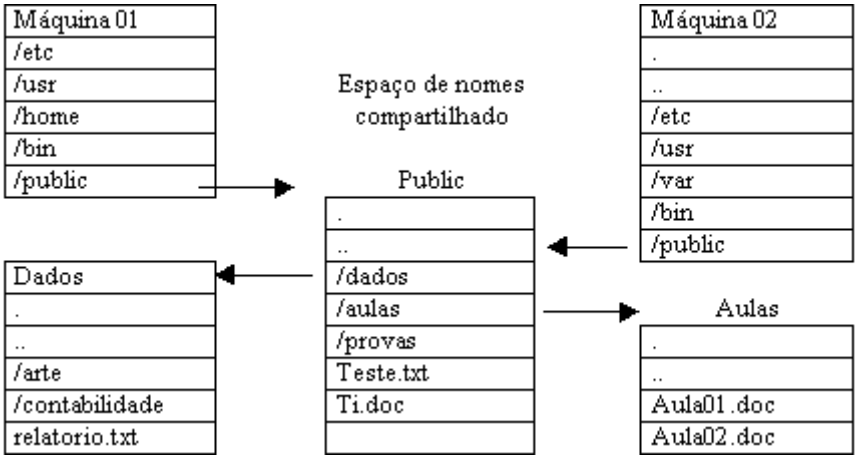


Figura 3.5 – Espaço de nomes do Amoeba.

Como o servidor de diretório é um componente crítico do sistema Amoeba, ele foi implementado para ser tolerante a falhas. A estrutura básica do servidor de diretório é um vetor de pares de capacidades que identificam objetos idênticos armazenados em servidores de arquivos distintos. O vetor é fisicamente armazenado em uma partição de disco própria do servidor de diretório e é 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 [TAN 92]. Esse esquema garante um alto grau de tolerância a falhas, porém, apresenta um custo relativamente alto, tanto em termos de ocupação de disco quanto em tempo de execução [FRO 94].

Servidor de Replicação

Os objetos gerenciados pelo servidor de diretório 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 várias réplicas do objeto [TAN 92].

O servidor de replicação é mantido em execução durante todo o tempo, verificando, periodicamente, partes do sistema de diretório. Sempre que uma entrada de diretório, suposta a ter n capacidades, é encontrada com menos capacidades, o servidor de replicação contata os servidores envolvidos para providenciar a geração das réplicas [TAN 92].

3.3.4 SPRITE

O Sprite é um sistema operacional distribuído desenvolvido pela Universidade da Califórnia, em Berkeley, que oferece um serviço de alta velocidade para redes locais e implementa um sistema de arquivos distribuído transparente. As principais características introduzidas no Sprite foram: a semântica UNIX em acessos concorrentes a arquivos distribuídos; a migração de processos de maneira transparente ao usuário, aproveitando os recursos computacionais de máquinas com pouca carga; uso de grandes *caches* de tamanho variável; resolução de nomes de caminho através de tabelas de prefixos; e, memória virtual implementada através de arquivos comuns [KON 96].

O núcleo do Sprite é *multithreaded,* o que significa que várias tarefas podem ser executadas concorrentemente dentro do *kernel*. A comunicação entre processos clientes e servidores é feita através de RPCs implementadas em nível de *kernel* [KON 96].

Espaço de nome

O espaço de nomes do Sprite é o mesmo para todos os clientes do sistema. Ele é composto por uma árvore de diretórios que é dividida em sub-árvores chamadas de domínios. Cada servidor é responsável por um ou mais domínios, como apresenta a figura 3.6. A resolução de nomes de caminhos para a localização física dos arquivos é totalmente transparente para o usuário, e é efetuada através de tabelas de prefixos dinâmicas, mantidas pelos clientes, que mapeiam domínios em servidores [KON 96].

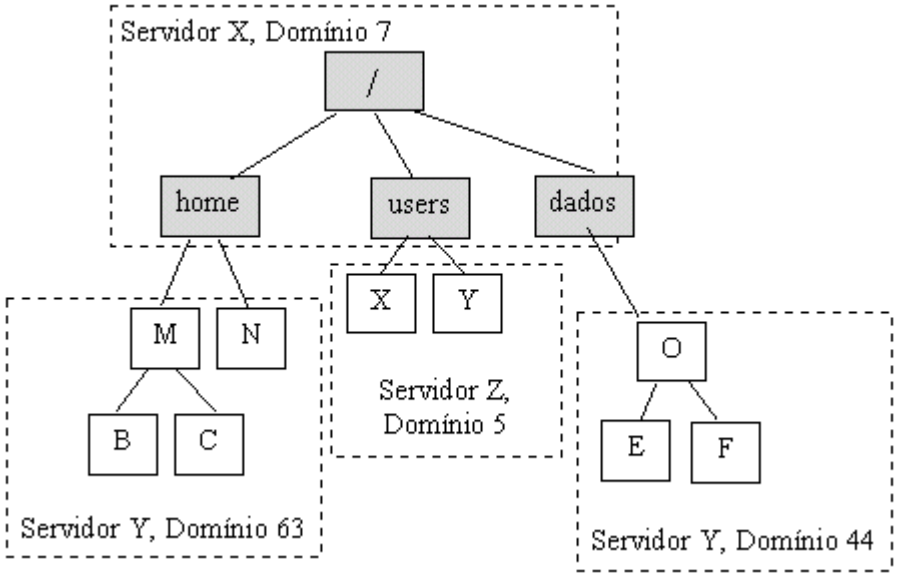


Figura 3.6 – Espaço de nomes dividido em domínios.

Quando um cliente necessita resolver um nome de caminho, ele verifica o prefixo mais extenso deste nome de caminho na tabela de prefixo local. Lá ele pode achar o servidor que manipula o arquivo ou diretório requisitado. O cliente, então, manda para o servidor apropriado o *token*, presente na terceira coluna da tabela, e o resto do nome de caminho. De posse dessas duas informações, o servidor pode tentar traduzir o resto do nome de caminho. Se o servidor consegue completar a tradução localmente, ele retorna para o cliente um manipulador de arquivo que é usado no acesso

subseqüente a este arquivo. Se o servidor não consegue resolver o nome de caminho requisitado, o cliente manda um *broadcast* para todos os servidores da rede, perguntando qual é o servidor responsável pelo arquivo procurado. Se o servidor responsável responde, o cliente adiciona uma entrada na sua tabela de prefixo local [KON 96]. A tabela de prefixo para o espaço de nomes da figura 3.6 é mostrada na figura 3.7.

Prefixo	**Servidor**	***Token***
/	X	7
/home/	Y	63
/users/	Z	5
/dados/	Y	44

Figura 3.7 – Tabela de prefixos.

Quando um cliente é inicializado, sua tabela de prefixos começa vazia. Em seguida, ele realiza um *broadcast* para descobrir quem é o responsável pelo domínio raiz. Assim como os processos clientes vão solicitando acesso a novos domínios, novos dados vão sendo acrescentados a tabela de prefixos. Se um cliente tentar acessar um domínio que está presente em sua tabela de prefixos, mas não receber resposta, ele invalida a entrada do domínio na tabela e executa um novo *broadcast* para todos os servidores, perguntando qual é o servidor para o domínio [KON 96].

O fato de um cliente ir montando sua tabela de prefixos, assim como vai acessando novos domínios, permite uma fácil implementação da migração de domínio. Se o disco de um servidor fica cheio, o administrador da rede apenas transfere um ou mais de seus domínios para outro servidor. Quando um domínio migra de um servidor para outro, os clientes automaticamente se adaptam às novas configurações [KON 96].

O NFS e o AFS resolvem nomes de caminho componente por componente. No Sprite, a tabela de prefixo permite o cliente acessar o servidor apropriado, mesmo que os servidores responsáveis pelos domínios do início do nome de caminho, estejam fora do ar [KON 96].

Cache

Como um dos objetivos do Sprite é a rapidez no acesso aos dados, e o acesso à memória é mais rápido do que o acesso ao disco local, os arquivos são armazenados temporariamente na memória *cache* do servidor. Isso permite a existência de clientes sem discos locais [KON 96].

Quando o Sprite foi desenvolvido, os dois principais objetivos dos projetistas eram aumentar o desempenho do Sprite e usar a semântica UNIX no compartilhamento de arquivos [KON 96]. Esses objetivos são descritos abaixo.

Aumentar o desempenho

Para aumentar o desempenho do sistema de arquivos Sprite foram adicionadas *caches* para tentar eliminar os acessos a disco, transações de rede, reduzir a carga da rede e dos servidores e aumentam a escalabilidade do sistema. Os clientes armazenam na memória *cache,* somente os blocos de dados de arquivos remotos, enquanto que os servidores armazenam blocos de dados, mapas dos arquivos dos discos físicos e outras informações que auxiliam na manutenção do serviço [KON 96].

Quando um processo cliente executa uma chamada de sistema *write*, os dados são escritos apenas nos blocos do arquivo que estão na *cache*. Depois de um determinado tempo são enviados ao servidor ou ao disco. Se o servidor, ou o cliente, sofre uma falha durante este período, os dados são perdidos. Para garantir a integridade dos dados, o Sprite oferece a possibilidade de forçar a gravação de dados para disco através do comando *fsync* do UNIX [KON 96].

Semântica UNIX

Outro objetivo do Sprite foi oferecer a semântica UNIX de acesso concorrente aos arquivos. Quando existem cópias de um arquivo no disco do servidor, na *cache* do servidor e nas *caches* de vários clientes, o Sprite garante a consistência entre todas essas cópias. A *cache* do cliente é desabilitada quando mais que um cliente tem um arquivo aberto, e pelo menos um desses clientes tem o arquivo aberto para escrita. Quando a

cache para um arquivo particular é desabilitada, todas as operações de escrita e leitura devem ser feitas diretamente no servidor, alcançando a semântica UNIX [KON 96].

Replicação

O Sprite não oferece nenhum tipo de replicação automática de domínios nos servidores. Logo, quando um servidor está fora do ar, não há como obter cópias dos seus arquivos nem alterá-los [KON 96].

Segurança

No Sprite, todos os clientes e servidores são considerados confiáveis, não havendo nenhum tipo de autenticação entre as máquinas, nem a possibilidade de criptografar informações [KON 96]. Quando um cliente deseja acessar um determinado arquivo, o Sprite não realiza necessariamente uma análise de cada componente do nome de caminho deste arquivo. Através da consulta à tabela de prefixos, é possível ir diretamente ao servidor responsável, o que elimina a possibilidade de verificação das permissões de acesso a cada componente do nome de caminho do arquivo. Em outras palavras, o Sprite não oferece a mesma segurança que o UNIX [KON 96].

3.3.5 Comparação dos Sistemas Apresentados

A tabela 3.1 apresenta, um resumo comparativo sobre as principais características dos sistemas de arquivos distribuídos estudados, analisando alguns aspectos considerados relevantes para este trabalho.

Tabela 3.1 - Comparação entre os sistemas de arquivos distribuídos estudados.

	Sun NFS	**ANDREW**	**AMOEBA**	**SPRITE**
Estado do servidor	Sem estado	Com estado	Sem estado	Com estado
Cache	Uso de memória principal para fazer cache no cliente e no servidor com objetivo de aumentar desempenho.	Uma partição no disco de cada cliente é usada como cache, onde são armazenadas as cópias dos arquivos.	Uso de *cache* no cliente e no servidor para armazenar arquivos inteiros na memória principal.	Uso de memória principal para realizar *cache* de arquivos, tanto no cliente como no servidor.
Replicação	Permite replicar informações somente para leitura.	Volumes são replicados apenas para leitura em vários servidores e atualizações são feitas na cópia original.	Facilitada com o uso de arquivos imutáveis. Quando um arquivo é criado, o servidor de replicação gera várias réplicas.	Não oferece replicação automática de domínios nos servidores.
Consistência	O servidor NFS é sem estado. Cliente fica responsável por verificar constantemente a consistência dos dados da *cache*.	Uso da semântica de sessão e do mecanismo de *callback* (promessa de ser a versão mais recente do arquivo).	Com o uso de arquivos imutáveis não há necessidade de mecanismos de sincronização.	Garante a consistência desabilitando a *cache* dos clientes e realizando todas as operações de leitura e escrita diretamente no servidor.
Escalabilidade	É considerado escalável, pois usa *cache* no cliente e no servidor e realiza a replicação de informação para leitura.	Para permitir maior escalabilidade, foi transferida uma grande quantidade de trabalho do servidor para o cliente. Arquivos são buscados no servidor e armazenados no disco local do cliente.	Sim, pois usa de replicação de arquivos imutáveis, distribuição de serviços em vários computadores, e suporta a migração de processos.	O uso de memória *cache* permite uma maior escalabilidade, pois mantém localmente uma cópia dos arquivos. Porém, para atualizar esses arquivos é necessário o uso de *broadcast*.
Desempenho	Para aumentar o desempenho o NFS usa memória *cache* no cliente e no servidor.	Tenta melhorar o desempenho com o armazenamento de arquivos na *cache* local do cliente.	Apresenta maior desempenho pois trabalha com arquivos imutáveis. Servidor mantém na memória principal uma tabela dos arquivos locais.	Tenta aumentar o desempenho implementando o servidor no nível do Kernel; usando *caches* no cliente e no servidor e realizando escritas retardadas do conteúdo dos arquivos.
Persistência de dados quando ocorre falhas	Escritas retardadas podem causar inconsistência e perda de dados.	Somente quando o arquivo é fechado ele é atualizado no servidor, podendo ocorrer a perda de dados.	Alto grau de tolerância a falhas, devido ao uso de replicação e por armazenar informações no disco.	Escritas retardadas podem causar perda de dados. Para garantir a consistência força a gravação de dados para disco.
Resolução de nomes	Feita componente por componente pelo cliente NFS	Feita componente por componente pelo cliente AFS (Venus)	Servidor de diretório é responsável por traduzir nomes de objetos em capacidades.	Resolução de nomes é feita usando tabela de prefixos, criada dinamicamente no cliente.
Espaço de nomes compartilhado	Cada cliente vê de forma diferente o espaço de nomes compartilhado.	Todos os clientes tem a mesma visão do espaço de nomes compartilhado.	Espaço de nomes compartilhado inicia no diretório */public*. Cada cliente vê um espaço de nomes próprio.	Todos os clientes tem a mesma visão do espaço de nomes compartilhado.
Segurança	Usa lista de controle de acesso, bits de proteção do UNIX, e algumas versões usam mecanismos de autenticação baseados em chave pública e criptografia.	Usa lista de controle de acesso, bits de proteção do UNIX, e algumas versões usam mecanismos de autenticação baseados em chave pública e criptografia.	Os arquivos (objetos) são identificados e protegidos por capacidades.	Clientes e servidores são considerados confiáveis. Não oferece a mesma segurança que o UNIX, pois elimina a verificação de permissões de acesso a cada componentes do nome.
Nós clientes sem disco	Monta um sistema de arquivos remoto no cliente, que é suportado por um servidor NFS remoto.	-	-	Uso de memória *cache* permite nós clientes sem disco.

4 O AMBIENTE DO *CLUSTER* CLUX

Este capítulo tem como objetivo apresentar o ambiente inicial que foi analisado e estudado, para permitir o projeto e a implementação da distribuição do sistema de arquivos. O ambiente é um *cluster* de computadores, chamado Clux, o qual já possui implementado um servidor de arquivos centralizado e uma interface de comunicação.

O *cluster* Clux foi definido para seguir os mesmos princípios do multicomputador Crux [COR 99]. Para conhecer um pouco mais sobre o multicomputador Crux, na seção 4.1 é feita uma breve descrição desse. Na seção 4.2 é apresentada a arquitetura do *cluster* Clux. A seção 4.3 descreve brevemente algumas características que levaram a escolha do sistema operacional Linux como plataforma de trabalho. A seção 4.3 descreve o servidor de arquivos centralizado e a seção 4.4 a interface de comunicação.

4.1 Multicomputador CRUX

O Crux é um multicomputador onde os nós são interconectados por uma rede *crossbar* e um barramento de controle. Como apresentado na figura 4.1, os componentes do multicomputador Crux são [COR 99]: os nós de trabalho, o nó de controle, a rede de trabalho e a rede de controle.

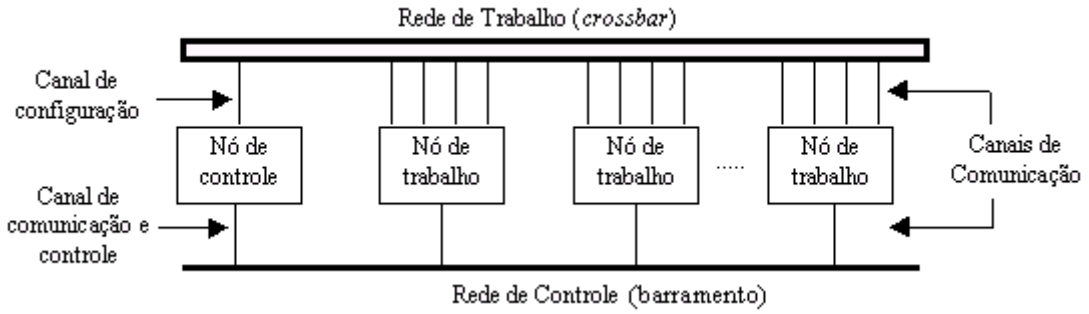


Figura 4.1 – Arquitetura do multicomputador Crux.

Os nós de trabalho constituem a maioria das máquinas existentes no *cluster* e realizam a execução das requisições por serviços, passadas para o *cluster*. Eles são computadores completos com processador, memória e várias interfaces de rede. As interfaces conectam os computadores a duas redes distintas: uma interface conecta a rede de controle, e uma ou mais, conectam a rede de trabalho.

O nó de controle é responsável por alocar e liberar os nós de trabalho, coordenando as comunicações entre os nós de trabalho do sistema, fazendo com que todas as trocas de mensagens entre eles sejam diretas.

A rede de trabalho é definida como uma rede de alta velocidade (*crossbar*) interligando suas máquinas. Ela é configurada pelo nó de controle e responsável pelo transporte de mensagens de tamanho arbitrário através de canais físicos diretos que são criados pela rede de controle, entre dois nós de trabalho quaisquer.

A rede de controle é usada para transportar mensagens de controle de tamanho pequeno, entre os nós de trabalho e o nó de controle, para a gerência das conexões dos canais diretos. Portanto, ela não necessita de uma rede de alto desempenho.

A arquitetura apresentada na figura 4.1 pode possuir um número expressivo de nós, cada um ligado à rede de trabalho por 4 canais de comunicação, podendo existir quatro canais físicos diretos conectados e trocando mensagens ao mesmo tempo. Além disso, possui um canal de comunicação com a rede de controle.

As conexões dos canais físicos só ocorrem se dois nós envolvidos desejarem a conexão, ou seja, os dois lados devem estar dispostos a trocar mensagens. Os nós de trabalho se comunicam através da rede de trabalho, através de canais físicos diretos. Antes do início da troca de mensagens é verificada a existência de um canal físico direto entre os nós trabalhadores. Caso exista um canal, a comunicação pode ser efetivada imediatamente; caso contrário, um pedido de conexão é enviado ao nó de controle, que cria os canais diretos que serão utilizados pelos nós de trabalho, enviando as mensagens de controle pela rede de controle. O nó de controle também recebe dos nós trabalhadores os pedidos de desconexão.

No Crux, cada nó de trabalho possui informações sobre as suas conexões, e o nó de controle que deve estar permanentemente acessível aos demais nós, possui informação sobre a situação de todos os nós do multicomputador.

4.2 Arquitetura do *Cluster* Clux

A figura 4.2 apresenta a arquitetura idealizada para o *cluster* Clux, que segue os mesmos princípios do multicomputador Crux [COR 99], descrito na seção 4.1.

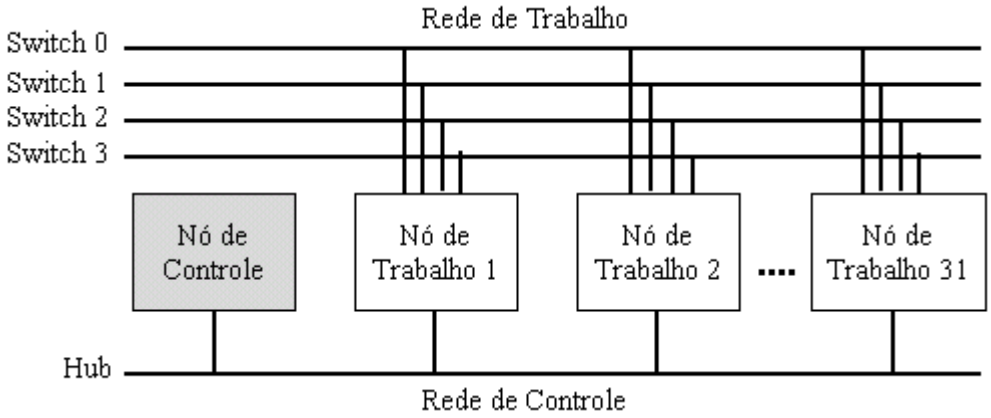


Figura 4.2 – Arquitetura idealizada para o *cluster* Clux.

Na figura 4.2, o *cluster* é composto por 32 microcomputadores completos, 4 *switches* e 1 *hub*. Desses microcomputadores, 31 são configurados como nós de trabalho e um como nó de controle. Os nós de trabalho possuem 5 placas *ethernet*, 1 conectada ao *hub* e 4 conectadas aos *switches*, e o nó de controle possui uma placa *ethernet* ligada ao *hub*. A ligação com o *hub* constitui a rede de controle e as ligações com os *switches* constituem a rede de trabalho. Esse ambiente idealizado para o *cluster* na realidade não existe. Assim, foi definido um ambiente com proporções menores, que opera com o mesmo princípio de funcionamento. A figura 4.3 mostra o ambiente do *cluster* Clux.

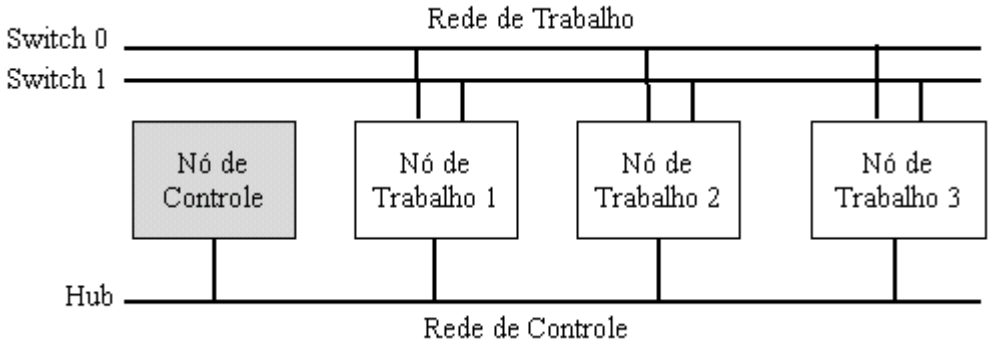


Figura 4.3 – Arquitetura real do *cluster* Clux.

Na figura 4.3, o ambiente montado para o desenvolvimento do *cluster* Clux é composto por 4 microcomputadores completos, 1 *hub* e 2 *switches*. Desses microcomputadores, 3 são configurados como nós de trabalho e 1 como nó de controle. Os nós de trabalho possuem 3 placas *ethernet*, 1 conectada ao hub e 2 conectadas aos *switches*, e o nó de controle possui uma placa *ethernet* ligada ao *hub*. A ligação com o *hub* constitui a rede de controle e as ligações com os *switches* constituem a rede de trabalho.

Para a rede de controle e para a rede de trabalho existem atualmente implementadas, duas interfaces de comunicação: a interface da rede de controle [BOG 02] e a interface da rede de trabalho [REC 02]. Porém, ambas as interfaces ainda não atendem as necessidades reais do *cluster* e necessitam ser remodeladas. Por definição, as interfaces devem estabelecer conexões e realizar a comunicação entre os processos do *cluster* Clux, usando *sockets*. Essas interfaces também implementam a idéia de nó de trabalho virtual, onde é simulado, num único nó de trabalho real, todo o ambiente idealizado para o *cluster* Clux, ou seja, um nó de trabalho real pode ser composto por 32 nós de trabalho virtuais, onde cada nó virtual executa somente um processo.

O *cluster* também conta com um servidor de arquivos centralizado, que recebe requisições clientes, efetua as operações requeridas e devolve o resultado. O sistema operacional usado como base para a implementação das aplicações do *cluster* é o Linux, o qual segue o modelo cliente/servidor [PLE 01].

4.3 O Linux como Sistema Operacional

O sistema operacional escolhido, como plataforma de trabalho para o projeto e a implementação do *cluster* Clux foi o Linux. Como o servidor de arquivos centralizado e as interfaces da rede de trabalho e da rede de controle, já haviam sido implementadas neste sistema operacional, continuou-se a utilizá-lo.

Pode-se dizer que existem várias características favoráveis ao uso desse sistema. Segundo Daines (2002), os 149 sistemas de computação mais rápidos no mundo são *clusters*. O terceiro supercomputador mais poderoso do mundo é um *cluster* com sistema operacional Linux. Na maioria das arquiteturas, os *clusters* com sistema

operacional Linux apresentam a habilidade de executar melhor que supercomputadores, por um significante baixo custo.

As características do sistema operacional Linux, interessantes para *clusters* são [FER 01]:

- **Estabilidade:** É baseado no sistema operacional UNIX, o qual é muito utilizado por administradores de sistemas, devido à sua estabilidade;
- **Eficiência:** Ele não requer um computador com grande capacidade de processamento para ser executado. Dependendo do objetivo que se queira atingir, até mesmo PC's mais antigos podem ser reaproveitados com grande eficiência usando-se este sistema operacional;
- **Sistema de rede:** Foi criado como um sistema de rede. Já que um *cluster* é uma rede de máquinas, sua utilização é direta;
- **Código aberto:** Possibilita que partes de sua estrutura sejam modificadas, visando uma melhor adaptação às características dos *clusters*;
- **Software livre:** Pode ser obtido pela Internet sem custo algum. Isto estimula a utilização e distribuição de software livre, e evita também problemas com licenças de software.

4.4 Servidor de Arquivos Centralizado

O servidor de arquivos centralizado foi projetado e implementado, para o *cluster*, em outra dissertação de mestrado [PLE 01]. A função desse servidor é oferecer aos processos clientes os serviços de acesso a informações armazenadas em disco.

Quando o servidor de arquivos centralizado foi implementado, a interface da rede de trabalho e a interface da rede de controle ainda não haviam sido implementadas. Essas interfaces gerenciam respectivamente as redes de trabalho e de controle do *cluster*. Desse modo, a interface da rede de trabalho, usada pelo servidor de arquivos, foi implementada provisoriamente com *sockets* TCP/IP.

A figura 4.4 apresenta o servidor de arquivos centralizado, que tem como principais componentes: os processos clientes, o processo servidor de arquivos, a

interface do sistema e a interface da rede de trabalho. Nas próximas seções são descritos esses componentes.

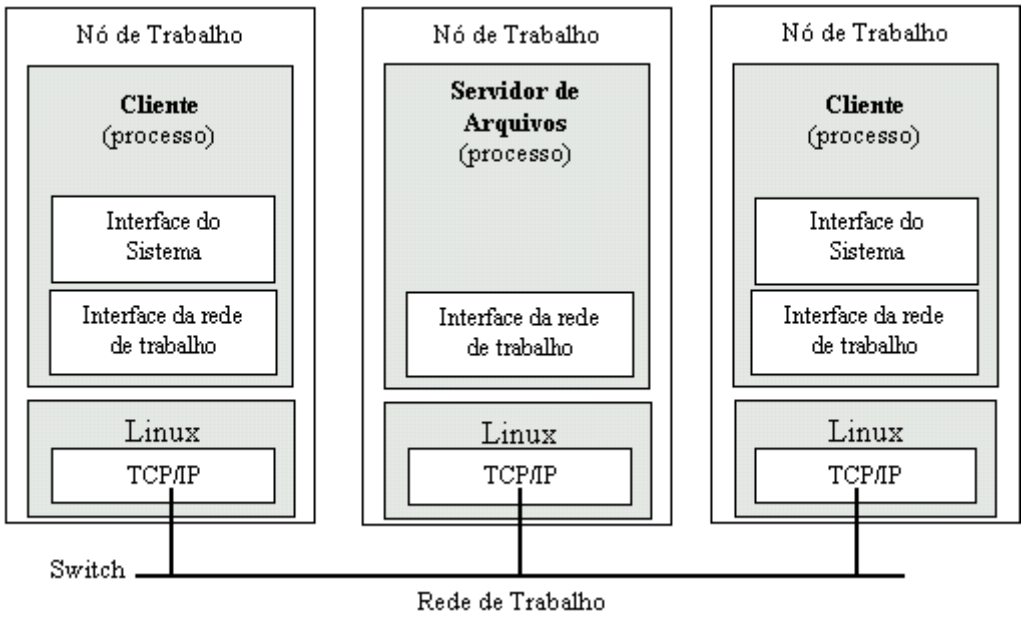


Figura 4.4 – Servidor de arquivos centralizado. Figura retirada da dissertação de mestrado de PLENTZ [PLE 01].

4.4.1 Processos Clientes

Quando o processo cliente executa uma chamada de sistema, ele utiliza, para a maioria das chamadas de sistema relacionadas ao sistema de arquivos, a biblioteca *libcsa*. Essa biblioteca implementa grande parte das chamadas de sistema relativas ao sistema de arquivos do Linux, e é representada na figura 4.4 pela interface do sistema.

4.4.2 Interface do Sistema

A interface do sistema foi implementada para os processos clientes pela biblioteca *libcsa,* que substituiu parcialmente a biblioteca *libc* original do Linux. Os operadores dessa interface não são executados no nó de trabalho onde se localiza o processo cliente, mas empacotados em mensagens que são enviadas ao nó de trabalho que possui o servidor de arquivos.

As funções que representam as chamadas de sistema executadas pelos processos clientes e que foram implementadas na *libcsa* são: *open, close, read, write, link, unlink, symlink, truncate, lseek, ftruncate, chdir, rename, mkdir, rmdir, mknod, chown, dup, creat, access, stat, sync, chmod* e *utime*. A função de cada uma dessas chamadas é descrita na tabela 4.1. As outras chamadas de sistema que não foram implementadas nesta interface são executadas localmente a partir da biblioteca *libc* original do Linux.

Tabela 4.1 – Chamadas de sistema implementadas na *libcsa*.

access(nome, modo)	Verifica se o usuário tem permissão para acessar o arquivo no modo especificado.
chdir(nome)	Muda o nome do diretório de trabalho para *nome*.
chmod(nome, modo)	Muda as permissões de acesso de um arquivo.
chown(nome, owner, grp)	Muda as propriedades do usuário e do grupo do usuário para o arquivo.
close(fp)	Fecha um arquivo aberto.
creat(nome, modo)	Cria o arquivo *nome* com as permissões de acesso especificadas em *modo*. Se o arquivo já existir, tenta abrir o arquivo para as permissões especificadas em *modo.*
dup(fd)	Duplica o descritor do arquivo *fd*, criando outro descritor que também aponta para o mesmo arquivo.
ftruncate(fd, length)	Trunca o arquivo *fd* para o tamanho específico *length*.
link(nome, nome2)	Cria uma nova entrada de diretório *nome2* para um arquivo já existente *nome*.
lseek(fd, offset, whence)	Move o ponteiro de leitura-escrita do arquivo *fd* para outra posição indicada por *offset e whence.*.
mkdir(nome, modo)	Cria o diretório *nome* especificando as permissões de uso.
mknod(nome, mode, dev)	Cria a entrada de diretório *nome* (um nodo do sistema de arquivos), onde *modo* especifica as permissões de acesso e *dev* o tipo de nodo a ser criado.
open(nome, flags)	Tenta abrir o arquivo *nome* conforme as permissões de acesso especificadas em *flags*.
read(fd, buffer, nbytes)	Lê uma determinada quantidade de bytes do arquivo *fd* e os coloca num *buffer*.
rename(nome, nome2)	Altera o *nome* de um arquivo para *nome2*.
rmdir(pathname)	Remove o diretório *pathname* se vazio.
stat(nome, buffer)	Adquire os atributos do arquivo *nome* e coloca no *buffer*.
symlink(nome, nome2)	Cria um novo nome para um arquivo. Criar um link simbólico chamado *nome* que conterá dentro dele *nome2*.
sync()	Força a gravação de informações para o disco.
truncate(path, length)	Trunca o arquivo *path* para um tamanho *length* específico.
unlink(nome)	Remove a entrada de diretório *nome*.
utime(nome, buffer)	Retorna a data e a hora do último acesso e modificação do arquivo *nome*.
write(fd, buffer, nbytes)	Escreve *nbytes* de dados do *buffer* para o arquivo *fd*.

Para a execução das chamadas de sistema realizadas pelos clientes, os operadores da biblioteca *libcsa* interagem com o processo servidor de arquivos, através da troca de mensagens. Para cada chamada de sistema realizada por um processo cliente, é montada uma mensagem contendo a identificação da chamada de sistema e seus argumentos, a qual é enviada para o servidor de arquivos.

4.4.3 Servidor de Arquivos

O servidor de arquivos interage diretamente com a interface da rede de trabalho, desenvolvida provisoriamente com *sockets* TCP/IP, e usa a interface do sistema de arquivos local do Linux para atender as requisições dos clientes. Ele é implementado com processos regulares do Linux, executando em um único nó de trabalho.

O servidor de arquivos é baseado no modelo cliente/servidor. Desse modo, um cliente solicita a realização de um serviço, enviando uma mensagem ao servidor de arquivos. O cliente é suspenso enquanto o serviço é efetuado pelo servidor. Ao término da execução do serviço, os resultados são enviados ao cliente em uma mensagem. A partir daí, o cliente pode retomar sua execução.

4.4.4 Interface da Rede de Trabalho

A interface da rede de trabalho foi implementada provisoriamente com *sockets* TCP/IP [PLE 01]. Os operadores que compõem a interface provisória foram definidos em CORSO (1999) e são mostrados a seguir [PLE 01]:

- ***s_Send (proc, msg, length):*** Esse operador pode ser usado tanto por processos clientes quanto por processos servidores, para enviar ao processo *proc* a mensagem *msg* de tamanho *length*.
- ***s_Receive (proc, msg, length)*:** Esse operador é usado pelo processo cliente para receber do processo *proc* (do servidor) a mensagem *msg* de tamanho *lenght*.
- ***s_ReceiveAny (proc, msg, length):*** Esse operador é usado pelo processo servidor para receber de um processo cliente qualquer a mensagem *msg* de tamanho *lenght*.

Esses três operadores foram implementados para realizar a comunicação entre os processos clientes e o servidor de arquivos centralizado. Assim, um cliente executa o operador *s_Send* para enviar uma requisição de serviço ao servidor de arquivos, seguido de *s_Receive* para a recepção da resposta do servidor. O servidor de arquivos executa o operador *s_ReceiveAny* para a recepção de uma requisição de serviço, e *s_Send* para o envio da resposta ao cliente.

4.5 Interfaces de Comunicação

As interfaces de comunicação desenvolvidas especialmente para o *cluster* Clux são a interface da rede de trabalho e a interface da rede de controle. Elas foram projetadas e implementadas em outras dissertações de mestrado [REC 02, BOG 02].

Como o ambiente idealizado para o *cluster* na figura 4.2 não existe, as interfaces da rede de trabalho e da rede de controle foram implementadas simulando esse ambiente num único nó de trabalho.

Neste trabalho, inicialmente, objetivou-se adaptar o servidor de arquivos centralizado ao ambiente do *cluster* Clux, substituindo a interface da rede de trabalho, provisoriamente implementada com *sockets* TCP/IP, pelas interfaces da rede de trabalho e da rede de controle, ambas desenvolvidas para o *cluster*. Porém, depois de realizar vários testes, verificou-se que as interfaces não corresponderam às expectativas. Desse modo, obtou-se por utilizar a interface de comunicação já utilizada pelo servidor de arquivos centralizado, ou seja, *sockets* TCP/IP.

5 A DISTRIBUIÇÃO DO SISTEMA DE ARQUIVOS NO CLUX

Neste capítulo é apresentado o projeto e a implementação de uma abordagem distribuída do sistema de arquivos no ambiente do *cluster* Clux. A seção 5.1 apresenta o modelo arquitetural proposto para a distribuição do sistema de arquivos e a descrição de cada componente que compõe o modelo. Na seção 5.2 são apresentados os detalhes da implementação, apresentando as estruturas definidas, os algoritmos genéricos dos processos envolvidos, os operadores e as funções implementadas.

5.1 Modelo Arquitetural

O principal objetivo deste trabalho é distribuir o sistema de arquivos no *cluster*. O primeiro passo para distribuir o sistema de arquivos é disponibilizar o serviço de arquivos em vários nós de trabalho dedicados, onde cada nó, disponibiliza parte dos arquivos para serem compartilhados por todos os processos clientes do *cluster*.

Tendo o serviço de arquivos disponível em vários nós de trabalho, é necessário implementar algum serviço que permita identificar, de forma transparente para o cliente, no ambiente do *cluster,* qual dos servidores disponibiliza determinado arquivo. Para isso, propõe-se a criação de um servidor de diretório.

O modelo arquitetural proposto para a abordagem distribuída do sistema de arquivos no ambiente do *cluster* é apresentado na figura 5.1. O ambiente montado é composto por 4 microcomputadores completos, chamados de nós de trabalho, os quais são interligados por uma rede conectada a um *switch*, a qual representa a rede de trabalho. Os principais componentes dessa arquitetura são os processos clientes, a interface do sistema, a interface da rede de trabalho, os servidores de arquivos e o servidor de diretório. Cada um deles é descrito a seguir.

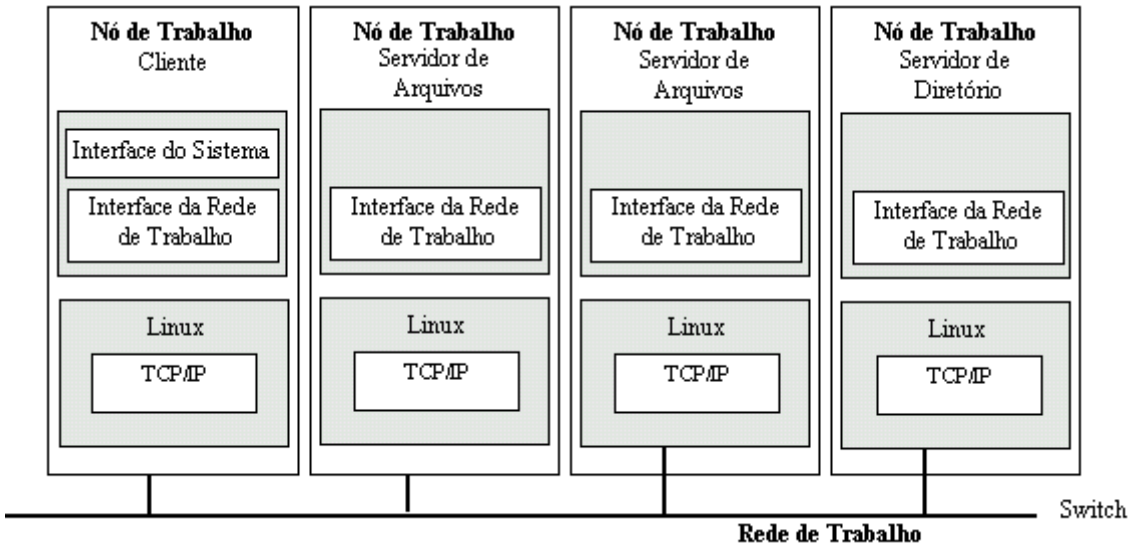


Figura 5.1 – Modelo distribuído do sistema de arquivos.

Processos clientes

A figura 5.1 mostra somente um processo cliente, porém, num ambiente distribuído podem existir vários processos clientes fazendo requisições por serviço ao mesmo tempo.

Os processos clientes utilizam, para a maioria das chamadas de sistema relacionadas ao sistema de arquivos, a biblioteca *libcsa.* Essa biblioteca é representada na figura 5.1 pela interface do sistema. Ela implementa grande parte das chamadas de sistema relativas ao sistema de arquivos do Linux.

Interface do Sistema

Na implementação do ambiente distribuído, a interface do sistema é implementada pela biblioteca *libcsa,* que substituiu parcialmente a biblioteca *libc* original do Linux. A interface é composta por um conjunto de operadores que representam algumas das chamadas de sistema executadas pelos processos clientes. As chamadas que foram implementadas na interface e que constituem a biblioteca *libcsa* são: *open, r_close, read, write, link, unlink, symlink, truncate, lseek, ftruncate, chdir, rename, mkdir, rmdir, mknod, chown, dup, creat, access, stat, sync, chmod* e *utime*. As outras chamadas de sistema que não foram implementadas nesta interface são executadas localmente a partir da biblioteca *libc* original do Linux. Além dos

operadores apresentados acima, existem também um conjunto de funções que são implementadas na interface do sistema e que são chamadas por esses operadores, como: *retorne_primeiro_nivel, localiza_servarq, traduz_fd, cria_fd, verifica_conexao, localiza_prefixo, verifica_conexao_2, inicializa_cliente_SD, inicializa_cliente_SA e localiza_SAs*. Essas funções são descritas na seção 5.2.3.

Para a execução das chamadas de sistema realizadas pelos clientes, os operadores e as funções da biblioteca *libcsa* interagem com os servidores de arquivos e com o servidor de diretório, através da troca de mensagens. Cada operador corresponde a uma função implementada na interface do sistema. Para cada chamada de sistema realizada por um processo cliente é montada uma mensagem, contendo a identificação da chamada e seus argumentos. Na realidade, a interface do sistema funciona como se fosse um *stub* cliente, recebendo a chamada de sistema, verificando seus argumentos e empacotando-os numa mensagem a ser enviada a um servidor. Todo nó de trabalho que roda um processo cliente necessita ter a interface do sistema.

Interface da Rede de Trabalho

Na arquitetura proposta, a comunicação entre processos clientes e servidores é realizada pela interface da rede de trabalho, que é implementada com *sockets* TCP/IP.

Os *sockets* são os mecanismos básicos da comunicação entre processos, atuando como um canal de comunicação, permitindo a um processo trocar mensagens com outro processo [COU 01]. Quando um *socket* é criado, um dos parâmetros especifica o protocolo a ser usado por ele em seus acessos à rede [TAN 97]. O protocolo IP é responsável pelo encaminhamento de pacotes de dados pelas diversas sub-redes, desde a origem até seu destino. O TCP (*Transmission Control Protocol*) é um protocolo da camada de transporte, que fornece um serviço de transporte confiável para a camada de aplicação [TAN 97]. Ele é responsável por recuperar dados corrompidos, perdidos, duplicados ou entregues fora de ordem. As conexões no TCP são ponto-a-ponto e transportam fluxos de dados em ambas as direções, caracterizando uma transmissão *full-duplex* [STE 98]. Os dados trocados pelas entidades emissoras e receptoras são na forma de segmentos [TAN 97].

O serviço TCP é obtido quando o transmissor e o receptor criam pontos terminais, denominados *sockets*. Esses pares de *sockets* identificam uma conexão TCP

de forma única na Internet. A figura 5.2 apresenta os passos da comunicação cliente/servidor usando *sockets* TCP/IP.

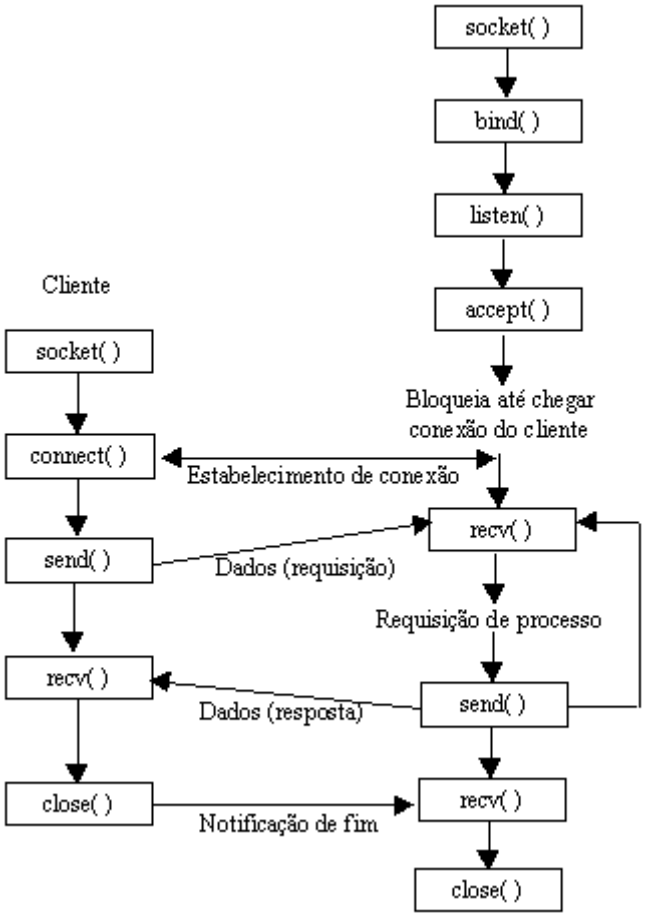


Figura 5.2 – Comunicação cliente/servidor usando *sockets* TCP/IP.

O processo cliente e o processo servidor usam, cada um, uma chamada de sistema *socket* para criar um descritor de *socket*, indicando o argumento SOCK_STREAM para especificar a comunicação TCP/IP.

No servidor, a chamada *bind* especifica um endereço local para o descritor do *socket*. Ele também usa a chamada *listen* para escutar seu descritor de *socket,* onde clientes requisitam conexões.

Quando os descritores de *sockets* são criados, tanto no computador fonte como no destino, pode então ser estabelecida uma conexão entre tais máquinas. O cliente requisita uma conexão com o servidor usando a chamada *connect,* que automaticamente liga o seu descritor de *socket* ao descritor de *socket* do servidor, colocando o *socket* no estado conectado [STE 98]. Em seguida, o servidor usa a chamada de sistema *accept* para aceitar uma requisição de conexão de um cliente e obter um novo *socket* para a

comunicação com aquele cliente. O *socket* original do servidor é usado para escutar outras requisições de conexão dos clientes [COU 01]. Devido a essa característica, o protocolo TCP é o mais utilizado para aplicações distribuídas baseadas no modelo cliente/servidor, porque vários clientes podem requisitar serviços ao mesmo tempo, o que requer que o servidor seja um servidor concorrente [JUN 96].

Depois que uma conexão tenha sido estabelecida, ambos os processos podem usar as chamadas de sistema *send* e *recv*, com seus respectivos descritores de *sockets,* para enviar e receber seqüências de bytes através da conexão. Quando a conexão não for mais necessária, ela pode ser desfeita através da chamada de sistema *close*. Quando um processo termina ou falha, todos os seus *sockets* são eventualmente fechados, e qualquer processo que tentar se comunicar com ele, descobrirá que sua conexão foi encerrada [TAN 97].

Devido às características acima e por facilitar a implementação do sistema, optou-se por utilizar *sockets* TCP/IP para realizar a comunicação no ambiente distribuído do sistema de arquivos.

Na interface da rede de trabalho, para realizar a troca de informações entre os processos, foram implementados três operadores: *s_Send, s_Receive* e *ss_ReceiveAny*. Esses operadores são descritos abaixo.

s_Send (sock, msg, lenght)

A função *s_Send* é usada para enviar mensagens de requisição de serviço dos processos clientes para os processos servidores e para enviar mensagens de resposta dos processos servidores para os processos clientes.

Na função, o argumento *sock* indica o descritor de *socket* a ser usado, o argumento *msg* fornece o endereço de memória onde estão os dados a serem enviados, e o argumento *lenght* especifica o número de bytes a serem enviados. Em caso de sucesso, a chamada retorna a quantidade de bytes enviados, caso contrário, retorna –1.

Essa função utiliza, na sua implementação, a chamada de sistema *send,* do protocolo TCP/IP, que é usada para enviar dados e funciona somente num *socket* conectado, pois não permite que o processo solicitante especifique um endereço de destino. O formato da chamada é: *send(fd, msg, len, flags).* Os três primeiros argumentos de *send* são equivalentes aos da chamada *s_Send*. O argumento *flags* é usado para controlar detalhes da comunicação e diagnosticar erros.

s_Receive (sock, msg, lenght)

A função *s_Receive* é usada pelos processos clientes para receber mensagens de resposta dos processos servidores. Essas mensagens, geralmente, contêm as respostas das execuções dos servidores.

Na função, o argumento *sock* é o descritor do *socket* do qual os dados devem ser recebidos, o argumento *msg* fornece o endereço de memória no qual os dados devem ser colocados, e o argumento *lenght* especifica o comprimento do *buffer* em bytes. Essa função retorna a quantidade de bytes recebidos, ou –1 em caso de erro.

A chamada de sistema utilizada na implementação dessa função é *recv,* do protocolo TCP/IP, que é usada para receber dados através de um *socket* conectado. O formato da chamada é o seguinte: *recv(fd, buf, len, flags)*, onde os três primeiros argumentos são iguais à função *s_Receive*, e o argumento *flags* controla detalhes da comunicação.

ss_ReceiveAny (sock, msg, lenght)

A função *ss_ReceiveAny* é usada por processos servidores para receber mensagens de requisição de serviço, tanto de processos clientes, como de outros processos servidores.

Na função, os argumentos usados são os mesmos definidos para a função *s_Receive*. Essa função também retorna a quantidade de bytes recebidos, ou –1 em caso de erro. A chamada de sistema utilizada na implementação dessa função também é *recv,* do protocolo TCP/IP.

Servidor de Arquivos

Considerando um *cluster* de computadores, com somente um servidor de arquivos centralizado para atender todas as requisições clientes do *cluster*, quando a quantidade de requisições clientes é muito elevada, o servidor de arquivos acaba perdendo desempenho, porque o tempo gasto com leitura ou escrita em disco é bastante significativo.

Como o principal objetivo deste trabalho é distribuir o sistema de arquivos no ambiente do *cluster*, o serviço de arquivos foi disponibilizado, inicialmente, em dois nós de trabalho. Isso permite a distribuição da carga de requisições clientes, melhorando a

disponibilidade dos servidores, pois cada servidor é responsável por gerenciar uma parte do espaço de nomes compartilhado. Cada um dos servidores de arquivos constitui um processo, que faz a inicialização das estruturas de dados e fica em laço, aguardando que os processos clientes estabeleçam conexões. Para cada conexão estabelecida é criada uma *thread* que fica aguardando mensagens de requisição de serviço. Quando uma requisição de serviço chega, a *thread* desempacota a mensagem, identifica a chamada de sistema, executa a chamada correspondente e retorna uma mensagem de resposta ao processo cliente. O uso de *threads* ao invés de processos permite aumentar ainda mais o desempenho dos servidores, pois o tempo gasto na troca de contexto entre *threads* é menor que o tempo gasto para realizar a troca de contexto entre processos.

Assim, pode-se dizer que o servidor de arquivos é *multithreaded,* que interage diretamente com a interface da rede de trabalho, desenvolvida provisoriamente com *sockets* TCP/IP, usa a interface do sistema de arquivos local do Linux para atender as requisições dos processos clientes e é baseado no modelo cliente/servidor.

Servidor de Diretório

O principal objetivo em se implementar um servidor de diretório é permitir que um processo cliente, de qualquer nó de trabalho, realize operações sobre arquivos ou diretórios, que compõem o espaço de nomes compartilhado, sem que ele conheça a localização física dos mesmos. O processo cliente não precisa saber em qual dos nós de trabalho está localizado o arquivo que ele deseja manipular. Ele vê um único espaço de nomes, onde não há indícios da localização física dos arquivos. Assim, pode-se dizer que o sistema implementa transparência de localização para o cliente.

O servidor de diretório é um processo que se encontra localizado num único nó de trabalho. Ele conhece o espaço de nomes compartilhado somente até o primeiro nível da árvore hierárquica, mantendo na Estrutura de Diretório da memória principal os prefixos das entradas de diretório disponibilizadas por cada servidor de arquivos e seu IP. Isso é suficiente para determinar o servidor de arquivos que disponibiliza determinada informação. Como esse trabalho não objetiva implementar nenhum mecanismo de tolerância a falhas para o servidor de diretório, caso ele venha a falhar, a estrutura mantida na memória principal será perdida, sendo necessário reiniciar todos os servidores novamente.

O servidor de diretório constitui um processo, que faz a inicialização das estruturas de dados e fica em laço, aguardando que os processos clientes estabeleçam conexões. A comunicação é realizada usando *sockets* TCP/IP. Quando um processo cliente estabelece uma conexão, o servidor de diretório cria um novo *socket,* que é usado para ficar à espera de mensagens de requisição vindas do processo cliente, usando o operador *ss_ReceiveAny***.** Quando chega uma requisição de serviço, o servidor de diretório cria uma *thread* para atendê-la. A *thread* desempacota a mensagem, identifica a chamada de sistema, executa a chamada correspondente e retorna uma mensagem de resposta ao processo cliente. Em seguida, a conexão com o processo cliente é fechada e a *thread* tratadora de requisições é encerrada.

Deste modo diz-se, que o servidor de diretório também é um servidor *multithreaded*, baseado no modelo cliente/servidor, que realiza a comunicação através da interface da rede de trabalho, e implementa diversas funções para atender as requisições dos processos clientes.

5.2 Implementação

Esta seção apresenta os detalhes da implementação da distribuição do sistema de arquivos no ambiente do *cluster*, descrito na seção 5.1. A seção 5.2.1 descreve a formação do espaço de nomes compartilhado, o modo que ele é visto pelos processos clientes e a identificação dos servidores no ambiente do *cluster.* Na seção 5.2.2 são apresentadas as diversas estruturas usadas na implementação do sistema. Na seção 5.2.3 é descrita a interface do sistema, abordando as funções implementadas, o algoritmo genérico dos operadores da interface, e alguns exemplos de código em linguagem C. Na seção 5.2.4 e 5.2.5 são apresentados respectivamente os algoritmos genéricos do servidor de arquivos e do servidor de diretório, algumas figuras contendo o código das principais funções desses servidores e comentários desses códigos.

5.2.1 Espaço de Nomes Compartilhado

O espaço de nomes do ambiente do *cluster* pode ser visto como uma árvore hierárquica, composta pelos arquivos disponibilizados pelos dois nós de trabalho, que

contém o serviço de arquivos. O espaço de nomes pode ser visto e compartilhado por todos os clientes do *cluster*. A figura 5.3 apresenta um exemplo do espaço de nomes compartilhado, composto por arquivos disponibilizados por dois servidores.

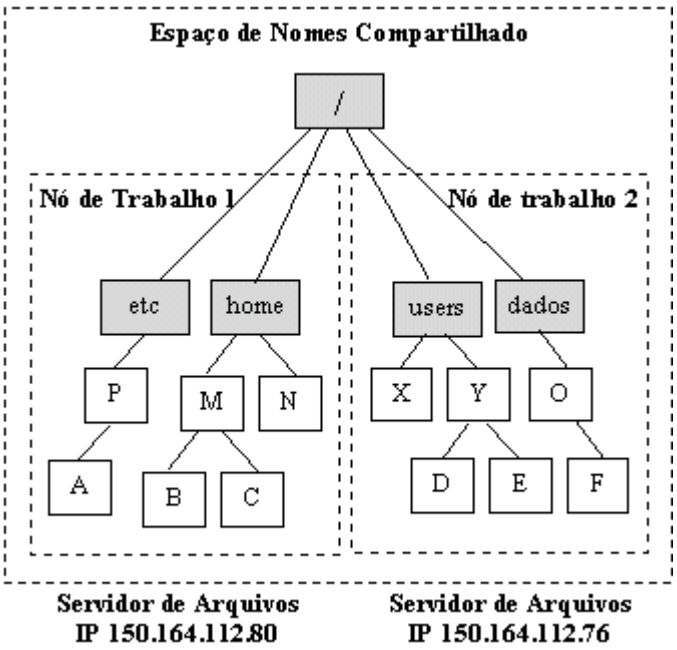


Figura 5.3 – Espaço de nomes compartilhado.

Para determinar quais diretórios de um nó de trabalho serão disponibilizados para compor o espaço de nomes compartilhado, o administrador do sistema cria um arquivo de configuração. Cada nó de trabalho que possui um servidor de arquivos, tem um arquivo de configuração armazenado no disco local. Nesse arquivo consta o número IP do nó de trabalho e os prefixos das entradas de diretório que ele está disponibilizando para o espaço de nomes compartilhado. Um prefixo representa o nome de caminho de um arquivo ou diretório, da raiz até o primeiro nível da árvore hierárquica. Por exemplo, o nome de caminho absoluto "*/etc/dados/teste.doc"* tem como prefixo "*/etc"*. Se o administrador do sistema mudar um servidor de arquivos para outro nó de trabalho, ele somente terá que redefinir as informações do arquivo de configuração.

O administrador do sistema deve tomar cuidado quando for criar os arquivos de configuração, para não colocar o mesmo prefixo em vários arquivos, pois o mesmo prefixo não pode estar presente em mais de um arquivo de configuração.

A figura 5.4 (a, b) mostra o conteúdo dos arquivos de configuração definidos para dois servidores de arquivos.

Figura 5.4 – (a) Servidor 1 e (b) servidor 2.

Nome de caminho dos arquivos

Os arquivos do espaço de nomes compartilhado não possuem nenhum tipo de identificador especial. Eles seguem a mesma nomenclatura usada pelo sistema de arquivos do Linux. Não existe nenhum tipo de diferenciação no nome de caminho para arquivos de outros nós de trabalho. Isso permite que a localização dos mesmos seja transparente para o cliente, ou seja, é como se todos os arquivos estivessem localizados no mesmo nó de trabalho. Por exemplo, "*/home/sistemas operacionais/Trabalho1.doc*" não apresenta indícios da localização física do arquivo no ambiente do *cluster*.

Sempre que uma chamada de sistema for realizada usando um nome de arquivo, esse nome deve ser o nome de caminho absoluto desse arquivo.

Identificação dos processos servidores

A definição de quais nós de trabalho do *cluster* vão executar os servidores de arquivos é feita pelo administrador do sistema.

Nos servidores, a identificação do serviço é feita pela definição de um número de porta para cada serviço. Por exemplo, o serviço de arquivos é realizado na porta 8001 e o serviço de diretório na porta 8002. Para identificar unicamente um servidor na rede, usa-se o número IP do nó de trabalho e o número da porta do serviço. A figura 5.5 apresenta um exemplo dessa identificação.

N.º IP do Nó de Trabalho	Número da Porta
150.164.112.76	8001

Figura 5.5 – Identificação de um processo servidor no *cluster*.

5.2.2 Descrição das Estruturas

As estruturas implementadas e usadas pelos diversos processo do *cluster* são apresentadas nessa seção.

Estrutura da mensagem

As mensagens trocadas entre os processos clientes e os processos servidores são formadas por um cabeçalho, comum a todas as chamadas de sistema, e um campo variável, específico para cada uma das chamadas de sistema. A figura 5.6 mostra o formato da mensagem.

No cabeçalho da mensagem o campo *process_id* identifica o processo cliente, o campo *call_id* identifica a chamada de sistema e o campo *size* identifica o tamanho da mensagem.

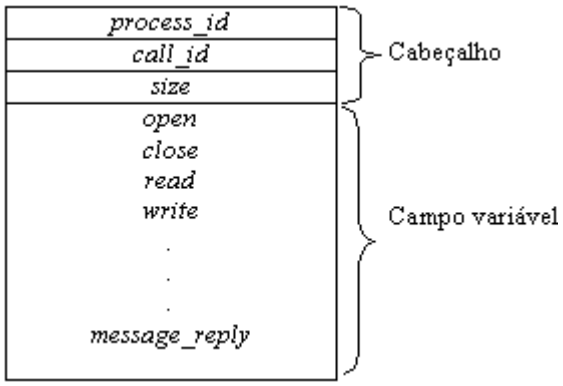


Figura 5.6 – Formato da mensagem.

Os campos *process_id* e *size* não são utilizados nesta implementação, mas foram mantidos no cabeçalho da mensagem por serem necessários a outros trabalhos, por exemplo, na interface de comunicação específica do *cluster* Clux.

O campo variável das chamadas de sistema é uma estrutura definida de acordo com os argumentos de cada chamada. A figura 5.7 mostra, como exemplo, uma mensagem que contém as informações referentes à chamada de sistema *open*, como o identificador da chamada e seus argumentos.

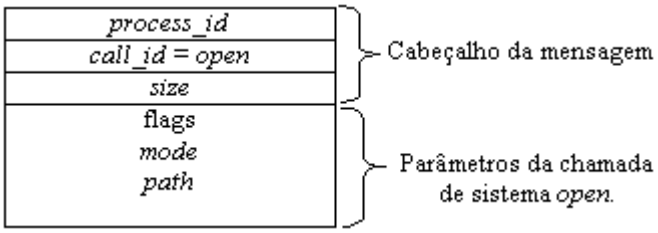


Figura 5.7 – Mensagem correspondente à chamada de sistema *open*.

A declaração da estrutura correspondente à chamada de sistema *open* é definida no arquivo *Tipos.h* e mostrada na figura 5.8. Assim, como foi definido para essa chamada, existem estruturas definidas para todas as chamadas de sistema implementadas pela biblioteca *libcsa*.

```
typedef struct
{
      int flags;
      int mode;
      char path[M_OPEN];
}s_open;
```

Figura 5.8 – Estrutura da chamada de sistema *open* definida na linguagem C.

A estrutura genérica de qualquer mensagem é definida no arquivo *Tipos.h* e é apresentada na figura 5.9.

```
typedef struct
{
      header headr;
      union
      {
            s_open            sopen;
            s_close           sclose;
            s_creat           screat;
            s_read            sread;
            s_lseek           slseek;
                  .
                  .
                  .
            s_teste101  steste101;
            s_teste103  steste103;
            message_reply     m_reply;
      } u_call;
} message;
```

Figura 5.9 – Estrutura da mensagem.

A estrutura da figura 5.9 é composta por um cabeçalho e por uma das estruturas definidas dentro de *union.* As estruturas *sopen*, *sclose*, *screat* e todas as outras, que foram definidas para cada chamada de sistema implementada na interface do sistema, vão receber os argumentos das chamadas de sistema realizadas pelo cliente. As estruturas *steste101* e *steste103* são usadas para manter as informações das mensagens de requisição enviadas dos processos clientes para o servidor de diretório.

Como parte do campo variável, existe ainda uma estrutura que corresponde às mensagens de retorno que são enviadas dos processos servidores aos processos clientes. Essa estrutura contém o resultado da execução da chamada de sistema solicitada ao processo servidor. Em caso de erro na execução das chamadas de sistema, o valor da variável global *errno* também é colocado nessa estrutura.

Na figura 5.10 é apresentada a estrutura da mensagem de retorno, em linguagem C, que foi definida no arquivo *Tipos.h.*

```
typedef struct
{
      int   result;
      int   m_errno;
      union
      {
            char buffer_reply[M_READ];
            struct stat buffer_stat;
            struct utimbuf  buffer_utime;
      } u_reply;
}message_reply;
```

Figura 5.10 – Estrutura da mensagem de retorno definida na linguagem C.

A estrutura da figura 5.10, também possui um *buffer* para conter os dados lidos na chamada de sistema *read*, e estruturas específicas utilizadas para conter os dados das chamadas de sistema *stat* e *utime.*

Estrutura da *Cache* Local do Cliente

A estrutura da *Cache* Local do Cliente é definida quando o processo cliente é inicializado. Essa estrutura é mantida na memória principal e tem como objetivo diminuir a quantidade de conexões estabelecidas entre o processo cliente e o servidor de diretório. Cada processo cliente tem uma *Cache* Local.

Cache Local do Cliente	
Prefixo	**IP**
/home	150.164.112.80
/etc	150.164.112.80
/dados	150.164.112.76

Figura 5.11 – *Cache* Local do Cliente.

Como mostra a figura 5.11, a *Cache* Local do Cliente possui dois campos: o prefixo, que indica o nome do caminho desde a raiz até o primeiro nível da árvore hierárquica, e o número IP do servidor de arquivos que está disponibilizando o prefixo.

As funções que foram implementadas para gerenciar a *Cache* Local do Cliente são: *cria_estrutura, insere_entrada, localiza_entrada* e *imprime_estrutura*. Elas foram implementadas no arquivo *Funções.h* e são descritas a seguir nessa seção.

Como se observa, não existe nenhuma função para excluir as entradas da estrutura de *Cache* Local do Cliente, pois foi definido que o primeiro nível da árvore hierárquica do espaço de nomes não poderá ser modificado enquanto o sistema estiver ativo. Assim, as informações mantidas na estrutura da *Cache* Local do Cliente não sofrerão modificações.

Caso o processo cliente venha a terminar, essa estrutura será excluída, pois subentende-se que os processos automaticamente desalocam os recursos alocados quando terminam. Quando o processo cliente inicia novamente sua execução, a estrutura de *Cache* Local do Cliente será novamente criada, com a função *cria_estrutura*.

Portanto, a estrutura *Cache* Local do Cliente é um tipo de *cache* que evita conexões e a troca de informações desnecessárias na rede. Isso ajuda a aumentar o desempenho do sistema, porque reduz a carga da rede e do servidor de diretórios, principalmente se a quantidade de requisições a esse servidor for muito grande. Também possibilita aumentar a escalabilidade do sistema.

Estrutura de Diretório

Como mostra a figura 5.4, cada servidor de arquivos possui um arquivo de configuração onde consta seu número IP e os prefixos que disponibiliza. A Estrutura de

Diretório é criada no servidor de diretório a partir das informações desses arquivos. Ela mantém na memória principal o número IP de cada servidor de arquivos e os prefixos que cada um está disponibilizando para serem compartilhados pelos processos clientes do *cluster*.

A figura 5.3 ilustra a composição do espaço de nomes compartilhado e a figura 5.12 a Estrutura de Diretório mantida na memória principal do servidor de diretório.

Estrutura de Diretório	
IP do Servidor de Arquivos	**Prefixo**
150.164.112.80	/etc
150.164.112.80	/home
150.164.112.76	/users
150.164.112.76	/dados

Figura 5.12 – Estrutura de Diretório.

A estrutura da figura 5.12 possui dois campos: o prefixo, que indica o nome da entrada de diretório somente até o primeiro nível da árvore hierárquica, e o número IP do servidor de arquivos que está disponibilizando essa entrada. Por exemplo, o servidor de arquivos com número IP 150.164.112.76 está disponibilizando para o espaço de nomes compartilhado os prefixos */etc* e */home.*

As informações mantidas na Estrutura de Diretório não sofrem modificações durante a execução do sistema, ou seja, não é permitida a modificação das informações até o primeiro nível do espaço de nomes compartilhado. Isso foi definido devido a um conjunto de fatores que viriam a dificultar a distribuição do sistema, que são:

- Se uma nova entrada de diretório fosse criada no primeiro nível do espaço de nomes compartilhado, antes de criá-la, seria necessário determinar em qual servidor de arquivos ela seria criada, verificando o espaço em disco. Depois de criada, seria necessário acrescentar o nome desta nova entrada na Estrutura de Diretório do servidor de diretório.
- Caso, uma entrada no primeiro nível do espaço de nomes compartilhado fosse excluída, seria necessário invalidar as informações da estrutura de *Cache* Local de cada processo cliente.
- Se uma entrada fosse renomeada, as informações da estrutura de *Cache* Local de cada processo cliente teriam que ser atualizadas.

- Seria necessário conhecer os prefixos já disponíveis e evitar que os novos arquivos ou diretórios criados no primeiro nível, tivessem nomes iguais aos prefixos já disponíveis. Por isso, decidiu-se não criar nem modificar as informações no primeiro nível da árvore durante a execução do sistema.

A Estrutura de Diretório é gerenciada por um conjunto de funções, que foram definidas no arquivo *Funcoes.h,* as quais são: *cria_estrutura, insere_entrada, localiza_entrada* e *imprime_estrutura.* Essas funções são descritas a seguir.

Funções usadas pela *Cache* Local do Cliente e pela Estrutura de Diretório

A Estrutura de Diretório definida no servidor de diretório e a estrutura de *Cache* Local do Cliente, mantêm o mesmo tipo de informação. Assim, ambas usam as mesmas funções, que foram implementadas no arquivo *Funcoes.h,* para gerenciar suas informações. Entre as funções tem-se:

- ***cria_estrutura:*** Função usada para criar uma estrutura vazia.
- ***insere_entrada:*** Função usada para inserir uma nova entrada na Estrutura de Diretório ou na *Cache* Local do Cliente. São passados o prefixo e o número IP do servidor de arquivos que disponibiliza esse prefixo. Caso a inserção seja realizada com sucesso retorna 1, senão retorna 0.
- ***localiza_entrada:*** Função usada para localizar na Estrutura de Diretório do servidor de diretório, ou na *Cache* Local do Cliente, a entrada correspondente ao prefixo passado como argumento. Se a localização for realizada com sucesso retorna 1 e o número IP do servidor de arquivos que disponibiliza esse prefixo, caso contrário retorna 0.
- ***imprime_estrutura:*** Função usada para imprimir a Estrutura de Diretório ou a estrutura de *Cache* Local do Cliente. A função simplesmente recebe um ponteiro que aponta para o início de uma das estruturas.

Tabela de Conexões do Cliente

Para cada processo cliente existe uma Tabela de Conexões, que indica com quais servidores de arquivos o cliente tem conexões abertas. Como existem dois servidores de

arquivos na arquitetura apresentada na figura 5.1, o cliente terá sempre, no máximo, duas conexões abertas, como mostra a figura 5.13. Essas conexões são encerradas quando o processo cliente termina.

Tabela de Conexões do Cliente	
IP	***Socket***
150.164.112.76	2
150.164.112.80	5

Figura 5.13 – Tabela de Conexões do cliente.

Foram definidas três funções para gerenciar a Tabela de Conexões do cliente. Essas funções são implementadas no arquivo *Funcoes.h* e são apresentadas abaixo:

- ***insere_entrada_conexoes**:* Essa função é chamada para inserir uma nova entrada na Tabela de Conexões do cliente, adicionando o IP do servidor de arquivos e o *socket* usado pelo cliente na conexão com o servidor. Retorna 1 se houve sucesso ou 0 se não foi possível inserir a nova entrada.
- ***localiza_entrada_conexoes:*** Função chamada para localizar uma entrada na Tabela de Conexões. É usado o IP para localizar a entrada. Localizando a entrada que contém o IP, é verificado o *socket* que está sendo usado na conexão com o servidor de arquivos. Se a entrada foi localizada com sucesso, retorna o *socket* e o valor 1. Se a entrada não foi encontrada, ou se a estrutura está vazia, retorna 0.
- ***imprime_estrutura_conexoes*:** Essa função é chamada para imprimir a Tabela de Conexões do cliente. É passado como argumento um ponteiro que aponta para o início da tabela. Em seguida, são impressas as informações mantidas na tabela.

5.2.3 Interface do Sistema

No modelo arquitetural descrito na seção 5.1, a interface do sistema é implementada pela biblioteca *libcsa*, que é utilizada pelos processos clientes para a maioria das chamadas de sistema relacionadas ao sistema de arquivos do Linux. Para entender melhor o funcionamento da interface do sistema são apresentadas a seguir as funções que foram implementadas na interface e que são usadas pelos operadores da

libcsa, assim como a descrição do algoritmo genérico das funções que representam os operadores da biblioteca *libcsa.*

Funções usadas pelos operadores da interface do sistema

Como visto na seção 5.1, a interface do sistema foi implementada por diversas funções, que são chamadas pelos operadores da *libcsa.* Cada uma dessas funções é descrita abaixo.

retorne_primeiro_nivel(nome, prefixo)

Função usada para extrair o prefixo do nome de caminho absoluto de um arquivo. O prefixo é o nome de caminho de um arquivo da raiz até o primeiro nível da árvore hierárquica.

localiza_servarq(sockfd_sd, prefixo, ip)

Essa função monta uma mensagem com o prefixo do nome do arquivo e envia para o servidor de diretório. O servidor de diretório verifica se o prefixo está na Estrutura de Diretório. Se estiver, retorna para o processo cliente o IP do servidor de arquivos que disponibiliza esse arquivo, ou 0 indicando que o prefixo não foi encontrado no servidor de diretório.

cria_fd(fd_global)

Para as chamadas de sistema que retornam um descritor de arquivos, como *open, creat* e *dup*, a interface do sistema usa a função *cria_fd* para criar um identificador único para um arquivo no ambiente do *cluster*, através da concatenação do descritor do arquivo com o descritor do *socket* que identifica uma conexão com o servidor de arquivos (*fd* + *socket*).

Esse identificador é criado considerando a possibilidade de um mesmo processo cliente abrir dois arquivos, de servidores diferentes, que tenham o mesmo descritor de arquivo. Para conseguir distinguir um do outro criou-se esse identificador único.

traduz_fd(fd_do_cliente, fd_global)

Essa função extrai do identificador único do arquivo o *fd* (descritor de arquivo) e o *socket* da conexão com o servidor.

verifica_conexao(nome)

Inicialmente, essa função chama a função *retorne_primeiro_nível,* para extrair o prefixo do nome de caminho absoluto do arquivo. Tendo o prefixo, a função *verifica_conexao* chamada à função *localiza_entrada* para verificar se esse prefixo se encontra na *Cache* Local do Cliente.

Se o prefixo não estiver na *Cache* Local do Cliente, a função *verifica_conexao* estabelece uma conexão com o servidor de diretório chamando a função *inicializa_cliente_SD.* Se a conexão for estabelecida com sucesso a função retorna o novo *socket.* Em seguida, chama-se a função *localiza_servarq*, que empacota o prefixo e o identificador do serviço (*m.headr.call = 103*) numa mensagem de requisição, que é enviada ao servidor de diretórios. O servidor de diretório recebe a mensagem, desempacota-a, e utiliza o prefixo para localizar na Estrutura de Diretório o número IP do servidor de arquivos. Se o prefixo e o IP correspondente forem encontrados, o número IP do servidor de arquivos é empacotado numa mensagem de resposta, que é enviada de volta para a função *verifica_conexao* na interface do sistema. Na interface do sistema, a função recebe a mensagem de resposta, com o IP do servidor de arquivos, e acrescenta o prefixo e seu IP na *Cache* Local do Cliente chamando a função *insere_entrada*. Se o prefixo não for encontrado na Estrutura de Diretório é retornada uma mensagem de erro para a função *verifica_conexao*, a qual atualiza a variável *errno* e retorna –1.

Se o prefixo for encontrado na *Cache* Local do Cliente, será possível determinar o número IP do servidor de arquivos que disponibiliza esse prefixo.

Conhecendo o número IP do servidor de arquivos, a função *verifica_conexao* verifica se já existe uma conexão estabelecida com esse servidor chamando a função *localiza_entrada_conexoes,* para localizar o número IP na Tabela de Conexões do cliente. Se o IP for encontrado na Tabela de Conexões do cliente, significa que o processo cliente já tem uma conexão estabelecida com esse servidor de arquivos, retornando o *socket* da conexão. Se o IP não for encontrado, significa que o processo

cliente não tem nenhuma conexão estabelecida com esse servidor de arquivos, havendo necessidade de estabelecer uma nova conexão.

Para estabelecer uma nova conexão é chamada a função *inicializa_cliente_SA* que passa como parâmetro o IP. Se a conexão for estabelecida com sucesso a função retorna o novo *socket.* Em seguida, é chamada a função *insere_entrada_conexoes* que vai inserir na Tabela de Conexões do cliente uma nova entrada, passando o número IP e o *socket* da conexão.

localiza_prefixo(prefixo, ip)

Essa função é chamada somente pela função *verifica_conexao_2*. A função *localiza_prefixo* chama a função *localiza_entrada* para ver se o prefixo do nome se encontra na *Cache* Local do Cliente. Se ele não estiver na *Cache* Local do Cliente é estabelecida uma conexão com o servidor de diretório, usando a função *inicializa_cliente_SD*. Em seguida é chamada a função *localiza_serarq,* que empacota o prefixo e o identificador do serviço (*m.headr.call = 103*) numa mensagem de requisição, que é enviada ao servidor de diretórios. O servidor de diretório recebe a mensagem, desempacota-a, e utiliza o prefixo para localizar na Estrutura de Diretório o número IP do servidor de arquivos. Encontrando o prefixo é retornado para a interface do sistema o IP do servidor de arquivos que disponibiliza esse prefixo. Em seguida o prefixo e o IP são inseridos na estrutura de *Cache* Local do Cliente, chamando a função *insere_entrada*. Caso o prefixo não se encontre na Estrutura de Diretório, a função *localiza_prefixo* retorna uma mensagem de erro e o processo cliente é encerrado.

verifica_conexao_2(nome1, nome2)

Para chamadas de sistema como *link, symlink, rename e chdir* que usam como argumentos dois nomes de caminho, é chamada a função *verifica_conexão_2*. Essa função chama a função *retorne_primeiro_nivel* para extrair o prefixo de cada um dos nomes de caminho. Em seguida, usa esses prefixos para descobrir se eles fazem referência a entradas de diretório disponibilizadas por um mesmo servidor de arquivos, ou se esses prefixos são disponibilizados por servidores diferentes. Para isso, é necessário descobrir o IP do servidor que disponibiliza cada prefixo, chamando a função *localiza_prefixo*.

Depois de identificar o IP de cada prefixo, compara-se os dois IPs para ver se são iguais. Se forem iguais, a função *localiza_entrada_conexoes* é chamada para localizar na Tabela de Conexões do cliente uma entrada com esse IP. Caso não exista, estabelece uma conexão com o servidor de arquivos, que tem esse IP, e em seguida acrescenta o IP e o *socket* da conexão estabelecida, na Tabela de Conexões do cliente. Ao final, a função *verifica_conexão_2* retorna o *socket* da conexão. Porém, se os números IPs forem diferentes, isso indica que não é possível realizar a operação desejada, porque os nomes de caminhos referem a arquivos de servidores diferentes. Assim, a variável *errno* é atualizada e a função retorna –1.

localiza_SAs(ips)

Essa função é chamada somente pelo operador *sync* da *libcsa*. Com essa função, a interface do sistema monta uma mensagem de requisição identificando o tipo de serviço (*mlocal.headr.call = 104*), estabelece uma conexão com o servidor de diretório usando a função *inicializa_cliente_SD*, e envia a mensagem ao servidor de diretório requisitando deste os números IPs de todos os servidores de arquivos ativos no *cluster*. Como resultado, o servidor de diretório retorna os IPs dos servidores, ou 0 em caso de erro. A função *localiza_SAs* retorna 0 em caso de erro, e 1 em caso de sucesso. No final, a conexão com o servidor de diretório é encerrada.

Em caso de sucesso, a interface do sistema irá conhecer os IPs de todos os servidores de arquivos ativos no *cluster*, estabelecendo uma conexão com todos eles e requisitando destes a execução da chamada de sistema *sync.*

Alteração do operador *close*

O operador *close* da biblioteca *libcsa* foi alterado para *r_close*, porque quando o processo cliente executava a chamada de sistema *close* para fechar uma conexão local, o identificador da chamada e seus argumentos eram empacotados numa mensagem e enviados a um dos servidores de arquivos remotos, para executar a operação. Assim, a conexão no cliente nunca era fechada.

Para resolver esse problema, foi definido que a chamada de sistema *close* será executada pela *libc* local do Linux e a chamada *r_close* será tratada pela biblioteca *libcsa.*

Algoritmo genérico dos operadores da biblioteca *libcsa*

Nesta seção é apresentada a figura 5.14, com o algoritmo genérico das funções que representam os operadores da biblioteca *libcsa,* e a descrição de seu funcionamento.

Linha 3 - Para cada chamada de sistema realizada por um processo cliente, é chamada a função correspondente na interface do sistema, que inicialmente verifica os possíveis erros relacionados aos argumentos utilizados.

Linha 4 – 6: Se algum argumento inválido for encontrado, a variável global *errno* é atualizada e o resultado é retornado imediatamente.

Linha 7 - 8 – Se os argumentos estiverem corretos, a interface do sistema chama a função *verifica_conexao*.

Linhas (9 – 12): Se a função *verifica_conexao* retornar –1 indica que o prefixo do arquivo não foi encontrado na *Cache* Local do Cliente, nem na Estrutura de Diretório do servidor de diretório.

Linhas (14 – 15): Se a função *verifica_conexao* retornar um *socket,* indica que o prefixo do nome do arquivo foi encontrado e que existe uma conexão estabelecida com o servidor de arquivos que disponibiliza esse arquivo.

```
1 Função(arg1, arg2, ...)
2 {
3  se existirem erros nos argumentos;
4  então
5     atualiza errno;
6     retorna erro;
7  senão
8     se verifica_conexão(nome) == -1
9     então
10          nome não foi encontrado;
11          conexão não foi estabelecida;
12          retorna erro;
13
14    nome foi encontrado;
15    conexão com SA foi estabelecida;
16    monta a mensagem;
17    envia a mensagem para SA;
18    espera mensagem de retorno do SA;
19    se o resultado retornado == -1
20    então
21          atualiza errno;
22
23    retorna resultado;
24 }
```

Figura 5.14 – Algoritmo genérico dos operadores da biblioteca *libcsa*.

Linhas (16 - 18): A interface do sistema empacota o identificador da chamada de sistema e seus argumentos numa mensagem de requisição e a envia para o servidor de arquivos. A execução do processo cliente é suspensa pela espera da mensagem de retorno do processo servidor. O servidor de arquivos recebe a mensagem de requisição, realiza o serviço requisitado e retorna uma mensagem de resposta para a interface do sistema.

Linhas (19 - 23): Na chegada da mensagem de retorno, se houver erros de execução, a variável global *errno* é atualizada. Em seguida, o resultado é retornado ao cliente.

Para exemplificar, a figura 5.15 apresenta o código do operador *open*, implementado na interface do sistema.

```
1 int open(const char *name, int flags, ...)
2 {
3     va_list ap;
4     int tam_msg, aux;
5
6     fd_arquivo_aberto call_fd;
7
8     if (strlen(name) > _POSIX_PATH_MAX) {
9                 errno =  ENAMETOOLONG;
10          return(-1);
11    }
12
13    va_start(ap, flags);
14    call_fd.socket = verifica_conexao(name);
15    if(call_fd.socket == -1){
16          return -1;
17    }
18
19    bzero(&m, sizeof(m));
20    // monta a msg a ser enviada ao servidor
21    strcpy(m.u_call.sopen.path, name);
22    m.u_call.sopen.flags = flags;
23    m.headr.call = 5;
24    m.headr.id = 1;
25
26    aux = va_arg(ap, int);
27    m.u_call.sopen.mode = aux;
28    va_end(ap);
29
30    tam_msg =   strlen(m.u_call.sopen.path)+
31                sizeof(m.u_call.sopen.flags)+
32                sizeof(m.u_call.sopen.mode)+ sizeof(m.headr);
```

Continua na próxima página

```
Continuação
33
34    s_Send(call_fd.socket, &m, tam_msg);
35
36    tam_msg =   sizeof(m.u_call.m_reply.result)+
37                sizeof(m.u_call.m_reply.m_errno)+
38                sizeof(m.headr);
39
40    s_Receive(call_fd.socket, &m, tam_msg);
41
42    if (m.u_call.m_reply.result == -1){
43          errno = m.u_call.m_reply.m_errno;
44          return -1;
45    }
46    else {
47          call_fd.fd = m.u_call.m_reply.result;
48          return cria_fd(&call_fd);
49    }
50 }
```

Figura 5.15 – Código do operador *open*.

5.2.4 Servidor de Arquivos

O algoritmo genérico do servidor de arquivos realiza a inicialização do servidor de arquivos. Após a inicialização, o servidor abre um arquivo de configuração local e estabelece uma conexão com o servidor de diretório. Esse arquivo contém o número IP do servidor de arquivos e os prefixos que ele está disponibilizando para o espaço de nomes compartilhado. O servidor de arquivos inicia o cadastro dessas informações na Estrutura de Diretório do servidor de diretório. Antes do servidor de arquivos cadastrar uma nova entrada na Estrutura de Diretório, ele deve verificar se essa entrada já existe na estrutura, usando a função *localiza_entrada.* Se a entrada não existir, chama a função *insere_entrada*, passando o prefixo e o IP.

Depois de realizar o cadastro, o servidor de arquivos criar um *socket listenfd*, atribuindo a ele um endereço e uma porta local (figura 5.16 (a)). Em seguida, um laço de repetição é executado, no qual o processo servidor de arquivos fica à espera de pedidos de conexões dos processos clientes. Quando chega um pedido de conexão, é criado um novo *socket connfd* (figura 5.16 (b)). Depois, o processo servidor cria uma *thread* para atender a conexão com esse cliente, passando para ela o *socket connfd* (figura 5.16 (c)). O servidor de arquivos fica usando o *socket listenfd* para escutar novas requisições clientes de conexão. Quando um novo pedido de conexão chega, um

novo *socket* é criado e outra *thread* tratadora de cliente é criada para atender essa nova conexão. O resultado final é mostrado na figura 5.16(d).

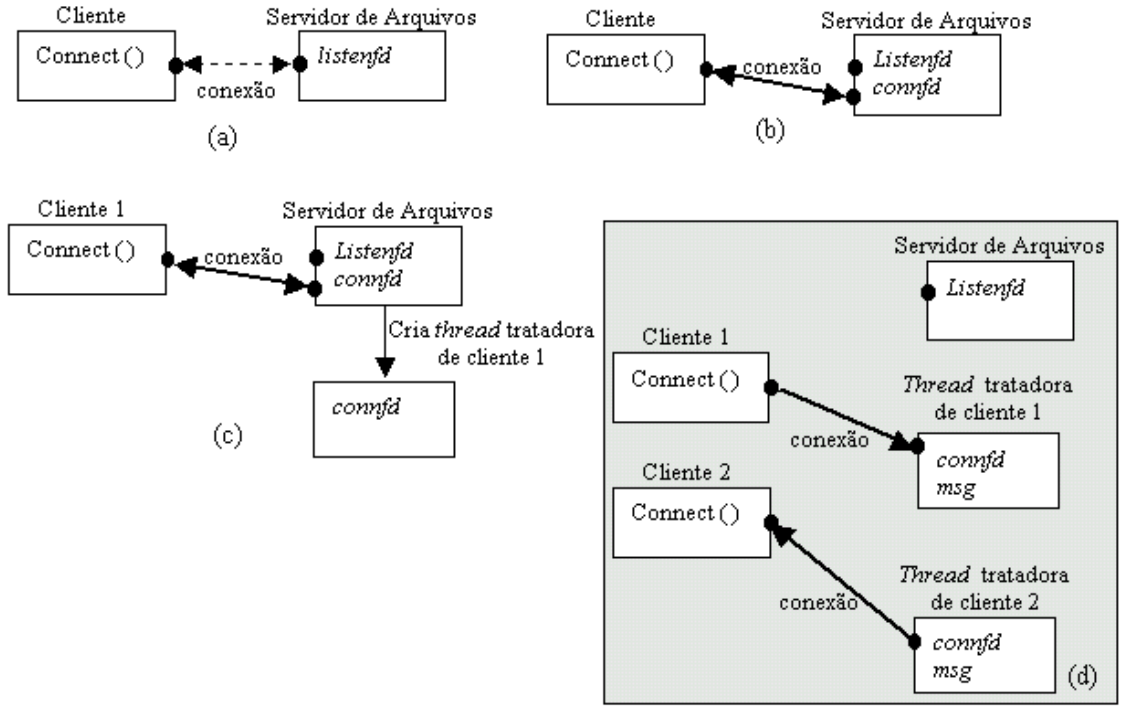


Figura 5.16 – Servidor de arquivos *multithreaded*, com *sockets* TCP/IP.

Cada *thread* tratadora de cliente executa um outro laço de repetição, no qual fica a espera de requisições desse cliente. Quando uma mensagem de requisição chega, a *thread* identifica, no cabeçalho da mensagem, a chamada de sistema a ser executada, desviando a execução do programa para o código associado àquela chamada. A chamada de sistema é então executada, e em seguida, são realizadas algumas verificações relacionadas ao valor de retorno da chamada. Depois, a mensagem de retorno ao processo cliente é montada e enviada. A *thread* tratadora de cliente somente é encerrada quando o cliente termina sua execução.

Detalhes da Implementação do servidor de arquivos

Para entender o funcionamento do servidor de arquivos, os códigos em linguagem C das principais funções são apresentados a seguir.

A figura 5.17 mostra o código da função *main* do servidor de arquivos, o qual é comentado logo abaixo.

```
1 main( )
2 {       pthread_t  th_id;
3         socklen_t clilen;
4         int connfd, listenfd, lenght;
5         struct sockaddr_in cliaddr;
6
7         printf("SA - Iniciando o servidor de arquivos.....");
8         cadastrar_servidor();
9         inicializa_servidor(&listenfd);
10        for ( ; ; )
11        {
12                clilen = sizeof(cliaddr);
13                if ((connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0)
14                {
15                        perror("Erro em accept");
16                        exit(1);
17                }
18                pthread_create(&th_id, NULL, thread_tratadora_de_cliente, (void *) connfd);
19        } //Fim do for
20 }// fim do main()
```

Figura 5.17 – Código da função *main* do servidor de arquivos.

Linhas 2 - 5: É realizada a definição de variáveis e estruturas usadas pelo servidor de arquivos.

Linha 8: O servidor de arquivos chama a função *cadastrar_servidor*, apresentada na figura 5.18, para estabelecer uma conexão com o servidor de diretório e cadastrar, na Estrutura de Diretório, os prefixos que o servidor de arquivos está disponibilizando para o espaço de nomes compartilhado, e seu IP.

Linha 9: A função *inicializa_servidor* cria um *socket* TCP com porta e endereço local, onde o servidor de arquivos aguarda a chegada de requisições de conexão dos processos clientes. Essa função é apresentada na figura 5.20.

Linhas 10 – 19: Um laço de repetição infinito é executado, onde o processo servidor de arquivos fica à espera de uma solicitação de conexão através da função *accept*. Na função *accept*, o argumento *listenfd* é o descritor do *socket*, o argumento *cliaddr* é um ponteiro para um endereço de memória onde ficará a estrutura do tipo *sockaddr*, e o argumento *clilen* indica o tamanho da estrutura. A função *accept* extrai a primeira requisição de conexão da fila de requisições e cria um novo *socket*, retornando o novo descritor do *socket connfd.* A estrutura *cliaddr* é preenchida com os dados do cliente que requer a conexão. O *socket* original *listenfd* permanece no estado de escuta por outras requisições de conexão. Porém, se o valor retornado da função *accept* indicar

erro de execução, uma mensagem é impressa e o processo servidor de arquivos é encerrado. O processo servidor mantém, então, dois *sockets* abertos, o original e o novo. Na linha 18 é chamada a função *pthread_create* que cria uma *thread*, que irá tratar da conexão estabelecida com o cliente. A *thread* criada chama a função *thread_tratadora_de_cliente* passando o *socket* da nova conexão estabelecida. A *thread* atenderá às requisições do processo cliente que chegam através do novo *socket.* Ao final, o laço de repetição é iniciado novamente. A função *thread_tratadora_de_cliente* é apresentada na figura 5.21.

A figura 5.18 mostra o código, em linguagem C, da função *cadastrar_servidor.* Essa função é executada pelo servidor de arquivos para abrir o arquivo *Entrada_sa1.txt*, ler as informações que foram determinadas pelo administrador do sistema nesse arquivo, estabelecer uma conexão com o servidor de diretórios, e enviar essas informações para serem inseridas na Estrutura de Diretório do servidor de diretório.

Linhas 3 – 7: Definição de variáveis e estruturas.

Linhas 9 - 13: A função *fopen* abre um arquivo chamado *Entrada_sa1.txt*. Se o valor retornado da abertura do arquivo indicar erro, uma mensagem de erro é impressa e o processo encerra sua execução.

Linha 14: No cabeçalho da mensagem existe o campo *call*, que identifica o tipo de serviço que o servidor de arquivos está requisitando do servidor de diretório.

Linha 15: É iniciada a execução de um laço *while,* onde o arquivo é lido, linha por linha, até chegar ao seu final.

Linha 17: A informação lida do arquivo é colocada na variável *buffer*.

Linhas 18 – 22: Quando a variável *i* for igual a 0 a variável *buffer* tem seu valor copiado para o campo *mm.u_call.steste101.addr* da mensagem, o qual recebe o endereço IP do servidor de arquivos.

Linhas 23 – 25: Quando *i* for maior que 0, indica que a informação lida para a variável *buffer* é um prefixo, o qual é copiado para o campo *mm.u_call.steste101.path* da mensagem.

Linha 26: É estabelecida uma conexão com o servidor de diretório, usando a função *inicializa_sa_cliente*, a qual retorna um *socket.* Essa função é apresentada na figura 5.19.

Linhas 27 - 28: Uma mensagem com o IP e o prefixo é enviada para o servidor de diretório usando a função *s_Send*. Na linha 28, o servidor de arquivos fica a espera de uma mensagem de resposta do servidor de diretório, executando a função *s_Receive*.

```
1 void cadastrar_servidor(void)
2 {
3       FILE *fp;
4       char buffer[257];
5       int tam, i=0;
6       int sockfd;
7       message mm;
8
9       if((fp  = fopen("Entrada_sa1.txt","r")) != NULL)
10              printf("\nSA - O arquivo <Entrada_sa1.txt> foi aberto.\n");
11      else {  perror ("Erro ao abrir o arquivo");
12              exit(1);
13      }
14      mm.headr.call = 101; //101 identifica a função insere_entrada() do servidor de diretório.
15      while(!feof(fp))
16      {
17              fscanf(fp,"%s\n",&buffer);
18              if(i==0)
19              {
20                      strcpy(mm.u_call.steste101.addr,buffer);
21                      i++;
22              }
23              else
24              {
25                      strcpy(mm.u_call.steste101.path,buffer);
26                      sockfd = inicializa_sa_cliente();
27                      s_Send(sockfd, &mm, tam);
28                      s_Receive(sockfd, &mm, tam);
29                      close(sockfd);
30                      if (mm.u_call.m_reply.result == 0)
31                      {
32                              perror("Erro no cadastro");
33                              exit(1);
34                      }
35                      else
36                              printf("\nSA - O SA se cadastrou no SD com sucessso.");
37              }
38      }
39      fclose(fp);
40 }
```

Figura 5.18 – Código da função *cadastrar_servidor*.

Linha 29: Depois de receber a mensagem de resposta, o servidor de arquivos encerra a conexão usando a função *close***.**

Linha 30: Se na mensagem de resposta o resultado retornado for igual a 0, indica que ocorreu um erro na realização do cadastro das informações no servidor de diretório. Assim, o servidor de arquivos imprime uma mensagem de erro e encerra sua execução. Caso contrário, o cadastro foi realizado com sucesso. Em seguida, o laço *while* é iniciado novamente.

Linha 39: Depois de terminar de cadastrar todas as informações no servidor de diretórios, o servidor de arquivos usa a função *fclose* para fechar o arquivo *Entrada_sa1.txt.*

A figura 5.19 apresenta o código, em linguagem C, da função *inicializa_sa_cliente*. Como visto na figura 5.18, a função *cadastrar_servidor* do servidor de arquivos chama a função *inicializa_sa_cliente* para estabelecer uma conexão com o servidor de diretório. O código da função é comentado logo abaixo.

```
1 int inicializa_sa_cliente(void)
2 {
3       int sockfd;
4       char ender[40] = "127.0.0.1"; // IP do SD
5       struct sockaddr_in serv_sa_addr;
6       if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
7       {
8               perror(" Erro na criação do socket");
9               exit(1);
10      }
11
12      bzero(&serv_sa_addr, sizeof(serv_sa_addr));
13      serv_sa_addr.sin_family = AF_INET;
14      serv_sa_addr.sin_port = htons(SERVDIR_PORT);
15      inet_pton(AF_INET, ender, &serv_sa_addr.sin_addr);
16
17      if ((connect(sockfd, (SA *) &serv_sa_addr, sizeof(serv_sa_addr))) < 0)
18      {
19              printf("\nSA - No inicializa_sa_cliente:\n");
20              perror("Erro em connect");
21              exit(1);
22      }
23      return sockfd;
24 }
```

Figura 5.19 – Código da função *inicializa_sa_cliente.*

Linhas 3 – 5: Tem-se a definição de variáveis e estruturas necessárias para estabelecer a conexão.

Linhas 6 – 10: A função *socket* é utilizada para a criação de um *socket* TCP. O argumento AF_INET especifica o protocolo usado. O argumento SOCK_STREAM especifica que o protocolo utilizado será o TCP. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o servidor de arquivos encerra a sua execução. Senão, um *socket* TCP é criado, o qual será usado para as interações com o servidor de diretório.

Linhas 12 – 15: A estrutura de endereçamento do *socket* deve ser preenchida com os valores adequados para o protocolo TCP.

Linhas 17 – 22: A função *connect* é utilizada para estabelecer uma conexão do servidor de arquivos com o servidor de diretórios. O argumento *sockfd* indica o descritor do *socket* a ser conectado. O argumento *serv_sa_addr* indica uma estrutura de *socket* na memória que contém o endereço de destino ao qual o *socket* deve ser vinculado. O último argumento indica o tamanho da estrutura do *socket*, em bytes, que se encontra na memória. Quando a chamada é executada com sucesso retorna 0. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o servidor de arquivos encerra sua execução.

Linhas 23 – 24: A função *inicializa_sa_cliente* encerra sua execução retornando o *socket* criado para a função *cadastrar_servidor.*

A figura 5.20 apresenta o código, em linguagem C, da função *inicializa_servidor().* Depois que o servidor de arquivos realizou o cadastro das informações na Estrutura de Diretório, ele executa a função *inicializa_servidor().* Essa função é usada para criar um *socket* que fica aguardando a chegada de mensagens de requisições de conexão dos processos clientes.

Linhas 1 – 8: Na linha 3 é definida uma estrutura de endereçamento de *socket* chamada *servaddr.* Da linha 4 até a linha 8, a chamada de sistema *socket* tenta criar um *socket* TCP. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o processo servidor de arquivos encerra a sua execução. Senão, um *socket* TCP é criado e usado para interagir com os processos clientes.

Linhas 10 – 13: A estrutura de endereçamento do *socket* deve ser preenchida com os valores adequados para o protocolo TCP.

Linhas 15 – 19: A função *bind* é utilizada pelo servidor de arquivos para atribuir um endereço local para ele. O argumento *listenfd* é o descritor do *socket* criado

previamente. O argumento *servaddr* é um endereço de memória da estrutura do *socket* que especifica o endereço local, ao qual o *socket* deve ser vinculado. O último argumento indica o tamanho da estrutura do *socket* na memória em bytes. A chamada *bind* retorna 0 em caso de sucesso. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o processo servidor de arquivos encerra sua execução.

```
1 void inicializa_servidor (int *listenfd)
2 {
3       struct sockaddr_in servaddr;
4       if ((*listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
5       {
6               perror("Erro na criação do socket");
7               exit(1);
8       }
9
10      bzero(&servaddr, sizeof(servaddr));
11      servaddr.sin_family      = AF_INET;
12      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
13      servaddr.sin_port        = htons(SERVARQ_PORT);
14
15      if (bind(*listenfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
16      {
17              perror("Erro na execução do bind");
18              exit(1);
19      }
20
21      if ((listen(*listenfd, LISTENQ)) < 0)
22      {
23              perror("Erro em listen");
24              exit(1);
25      }
26 }
```

Figura 5.20 – Código da função *inicializa_servidor*.

Linhas 21 – 26: A função *listen* é utilizada pelo processo servidor de arquivos para esperar, através do *socket,* as requisições de conexões dos processos clientes. O argumento *listenfd* é o descritor do *socket* usado pelo servidor para escutar as requisições. O argumento LISTENQ especifica o tamanho máximo da fila de requisições para aquele *socket*. Se a fila estiver cheia quando uma requisição chegar, essa é recusada e descartada. A função retorna 0 em caso de sucesso, ou –1 em caso de erro, imprimindo uma mensagem de erro e encerrando a execução do servidor.

Na figura 5.21 é apresentado o código, em linguagem C, da função *thread_tratadora_de_cliente.* Quando é estabelecida uma conexão com um processo cliente é criada uma *thread* na função *main,* que chama a função *thread_tratadora_de_cliente*, para tratar dessa conexão**.**

Linhas 3 – 8: A *thread* realizada a definição e inicialização de variáveis e estruturas necessárias para tratar da conexão com o processo cliente.

```
1 void *thread_tratadora_de_cliente(void *__args)
2 {
3     int connfd = (int) __args;
4     message mm;
5     sd_thread_args *th_args;
6     message *th_mesg = NULL;
7     int lenght, num_bytes;
8     int num = 0;
9
10    printf("\nSA - Thread tratadora de conexão: %d\n",pthread_self());
11    bzero(&mm, sizeof(mm));
12    num_bytes = ss_ReceiveAny(connfd, &mm, lenght);
13    while(num_bytes>0)
14    {
15          mm.headr.id = 0;
16          th_mesg = (message *) malloc(sizeof(mm));
17          memcpy(th_mesg,&mm,sizeof(mm));
18
19          // Cria e inicializa buffer para argumentos;
20          th_args=(sd_thread_args *) malloc(sizeof(sd_thread_args));
21          th_args->mesg = th_mesg;
22          th_args->connfd = connfd;
23          printf("\nSa-Tamanhos:len=%d sizeof(mm)=%d\n",lenght,sizeof(mm));
24          switch(mm.headr.call)
25          {
26         case 3: { re_read(th_args);   break; }
27         case 4: { re_write(th_args);  break; }
28         case 5: { re_open(th_args);   break; }
29         ...
30         case 106: { re_stat(th_args);break; }
31           default:
32           printf("\nSA - DELAULT: Erro no Switch: <mm.headr.call= %d > invalido\n",mm.headr.call);
33          }
34
35          free(th_args->mesg);
36          free(th_args);
37
38          bzero(&mm, sizeof(mm));
39          num_bytes = ss_ReceiveAny(connfd, &mm, lenght);
40      }
41
42      printf("\nSA - Fim da thread tratadora de conexao %d.\n",pthread_self());
43      close(connfd);
44      pthread_exit(NULL);
45      return NULL;
46 }
```

Figura 5.21 – Código da função *thread_tratadora_de_cliente*.

Linha 11: É realizada a inicialização da estrutura *mm* que receberá a mensagem de requisição do cliente.

Linha 12: A *thread* que trata da conexão com o cliente, fica à espera de mensagens de requisição através do operador *ss_ReceiveAny*. Na chegada de uma nova mensagem de requisição do cliente, é calculada a quantidade de bytes recebidos.

Linha 13: É iniciado o laço *while*, que será repetido enquanto a quantidade de bytes recebidos for maior que 0.

Linhas 14 – 23: Dentro do laço, a *thread* que trata da conexão com o cliente cria e inicializa as estruturas *th_mesg* e *th_args*. A estrutura *th_args* recebe a mensagem de requisição vinda do processo cliente e o *socket* da conexão estabelecida com o cliente.

Linhas 24 - 33: O operador *switch,* da linguagem C, verifica qual é a chamada de sistema solicitada pelo processo cliente, verificando o campo *mm.headr.call***,** e desvia a execução do programa para o código correspondente.

Linhas 35 – 36: Depois que o código da chamada de sistema requisitada pelo processo cliente é executado, o espaço alocado para a estrutura *th_args* é liberado.

Linhas 38 – 39: A estrutura *mm* é novamente inicializada e usada para receber a próxima mensagem de requisição do cliente. A *thread* que trata da conexão com o cliente, fica novamente à espera de mensagens de requisição através do operador *ss_ReceiveAny*. Na chegada de uma nova mensagem de requisição do cliente, é calculada a quantidade de bytes recebidos. Se a quantidade de bytes recebidos for maior que 0, o laço *while* é novamente iniciado.

Linhas 42 – 45: No momento que a quantidade de bytes recebidos pela *thread* que trata da conexão com o cliente for menor ou igual a 0, a conexão e a *thread* são finalizadas. A chamada de sistema *close* é invocada para encerrar a conexão com o processo cliente.

5.2.5 Servidor de Diretório

O algoritmo genérico do servidor de diretório realiza, inicialmente, a inicialização de estruturas e variáveis. Em seguida, é realizada a criação do *socket listenfd*, atribuindo um endereço local para ele, e a especificação de um comprimento de fila para acomodar as solicitações simultâneas que chegam ao servidor.

Após a etapa de inicialização, o processo servidor executa um laço de repetição infinito no qual fica à espera de conexões dos processos clientes, através do *socket listenfd* (figura 5.22 (a)). Quando chega uma requisição de conexão de um processo cliente, o servidor de diretório cria um novo *socket connfd,* estabelecendo uma conexão com esse cliente (figura 5.22 (b)). Em seguida, o servidor fica aguardando uma mensagem de requisição desse cliente através do comando *ss_ReceiveAny*. Quando uma mensagem de requisição chega, o servidor cria uma *thread* tratadora de requisição para atender essa requisição, passando para a *thread* o *socket connfd* e a mensagem recebida (figura 5.22 (c)). A *thread* tratadora de requisição identifica, no cabeçalho da mensagem, a operação a ser executada, desviando a execução do programa para o código associado àquela operação. A operação é então executada, e em seguida, são realizadas algumas verificações relacionadas ao valor de retorno da operação. Depois, a mensagem de retorno é montada e enviada para o processo cliente usando o operador *s_Send* (figura 5.22 (d)). Em seguida, a *thread* tratadora de requisições é encerrada.

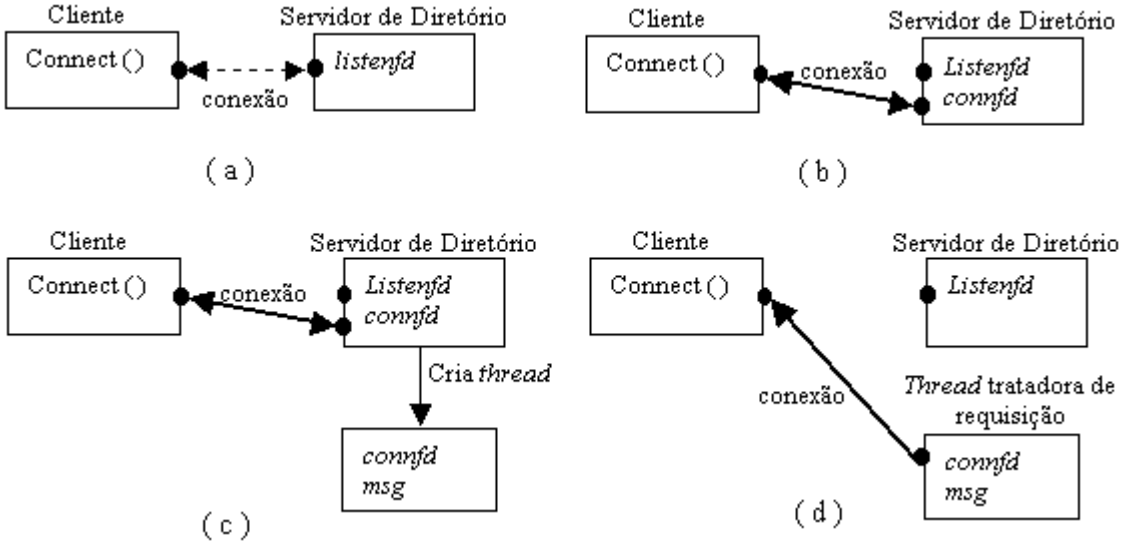


Figura 5.22 – Servidor de diretório *multithreaded*, com *sockets* TCP/IP.

Detalhes da Implementação do Servidor de Diretório

Para entender melhor o funcionamento do servidor de diretórios, algumas funções em linguagem C são apresentadas abaixo.

A figura 5.23 apresenta o código da função *main* do servidor de diretório, que é comentado logo abaixo.

Linhas 1 – 9: A função *main* do servidor de diretório inicia criando variáveis e estruturas.

Linha 12: O processo servidor de diretório executa a função *inicializa_servidor*, na qual um *socket* TCP é criado e o endereço e porta local são definidos. O código dessa função é apresentado na figura 5.24.

```
1 main()
2 {
3       message mm_d;
4       sd_thread_args *th_args;
5       message *th_mesg = NULL;
6       pthread_t  th_id;
7       socklen_t clilen;
8       struct sockaddr_in cliaddr;
9       int listenfd, lenght, connfd;
10
11      printf("\nSD - Iniciando o Servidor de Diretório.....\n");
12      inicializa_servidor(&listenfd);
13      cria_estrutura(&Estrutura);
14
15      for ( ; ; )
16      {
17              clilen = sizeof(cliaddr);
18              printf("\nSD - Antes de executar accept .....\n");
19              if ((connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0)
20              {
21                      perror("SD - Erro em accept");
22                      exit(1);
23              }
24
25              bzero(&mm_d, sizeof(mm_d));
26              ss_ReceiveAny(connfd, &mm_d, lenght);
27
28              // Cria buffer para mensagem
29              th_mesg = (message *) malloc(sizeof(mm_d));
30              memcpy(th_mesg,&mm_d,sizeof(mm_d));
31
32              // Cria e inicializa buffer para argumentos;
33              th_args = (sd_thread_args *) malloc(sizeof(sd_thread_args));
34              th_args->mesg = th_mesg;
35              th_args->connfd = connfd;
36
37              pthread_create(&th_id, NULL, thread_tratadora_de_requisicoes, th_args);
38              printf("\nSD - Thread criada: %d\n",th_id);
39   } //Fim do for
40 }// fim do main()
```

Figura 5.23 – Código da função *main* do servidor de diretório.

Linha 13: É executada a função *cria_estrutura* que cria e inicializa a Estrutura de Diretório.

Linhas 15 - 23: Um laço de repetição infinito é executado, onde o processo servidor de diretório fica à espera de uma solicitação de conexão através da função *accept*. Quando chega um pedido de conexão, se o valor retornado da função *accept* indicar erro de execução, uma mensagem é impressa e o processo servidor de arquivos é encerrado. Senão, a função *accept* retorna um novo *socket, connfd*. O processo servidor mantém então, dois *sockets* abertos, o original e o novo.

Linha 25: Em seguida, dentro do laço de repetição, a estrutura *mm_d* é inicializada como vazia.

Linha 26: Usando o operador *ss_ReceiveAny* e o novo *socket connfd*, o servidor de diretório fica à espera de mensagens de requisição do processo cliente.

Linhas 29 - 30: É alocado espaço de memória para uma estrutura chamada *th_mesg*. Depois, é copiado o conteúdo da estrutura *mm_d*, que contém a mensagem recebida, na linha 26, para a estrutura *th_mesg*.

Linhas 32 – 35: É criada uma estrutura chamada *th_args*, a qual conterá o *socket* que identifica a conexão com o cliente e a mensagem que o servidor de diretório recebeu do cliente através da conexão.

Linhas 37 – 40: A função *pthread_create* cria uma *thread* para tratar da mensagem de requisição cliente que chegou através da conexão estabelecida com o processo cliente. A *thread* criada, executa a função *thread_tratadora_de_requisicoes*, apresentada na figura 5.25, passando como parâmetro a estrutura *th_args* que contém o *socket* e a mensagem de requisição recebida. Enquanto a *thread* está executando a função *thread_tratadora_de_requisicoes,* o processo servidor de diretório imprime a mensagem de *thread* criada, e reinicia o laço de repetição novamente.

A figura 5.24 apresenta o código da função *inicializa_servidor*, que é comentado logo abaixo.

Linha 3: Definição da estrutura de endereçamento do *socket*.

Linhas 4 – 8: Na função *inicializa_servidor,* a função *socket* é utilizada para a criação de um *socket* TCP. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o processo servidor de diretório encerra a sua execução. Senão, um descritor de *socket* TCP é criado.

```
1 void inicializa_servidor(int *listenfd)
2 {
3       struct sockaddr_in serv_dir_addr;
4       if ((*listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
5       {
6               perror("Erro na criação do socket");
7               exit(1);
8       }
9
10      bzero(&serv_dir_addr, sizeof(serv_dir_addr));
11      serv_dir_addr.sin_family      = AF_INET;
12      serv_dir_addr.sin_addr.s_addr = htonl(INADDR_ANY);
13      serv_dir_addr.sin_port        = htons(SERVDIR_PORT);
14
15      if (bind(*listenfd, (SA *) &serv_dir_addr, sizeof(serv_dir_addr)) < 0)
16      {
17              perror("Erro na execução do bind");
18              exit(1);
19      }
20
21      if ((listen(*listenfd, LISTENQ)) < 0)
22      {
23              perror("Erro em listen");
24              exit(1);
25      }
26 }
```

Figura 5.24 - Código da função *inicializa_servidor*.

Linhas 10 – 13: A estrutura de endereçamento do *socket* deve ser preenchida com os valores adequados para o protocolo TCP.

Linhas 15 – 19: A função *bind* é utilizada pelo processo servidor de diretório para atribuir um endereço local para ele. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o processo servidor de diretório encerra a sua execução.

Linhas 21 – 26: A função *listen* é utilizada pelo processo servidor de diretório para esperar através do *socket listenfd* as requisições de conexões dos processos clientes. O argumento LISTENQ especifica o tamanho máximo da fila de requisições para aquele *socket*. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o processo servidor de diretório encerra a sua execução.

Na figura 5.25 é apresentado o código da função *thread_tratadora_de_requisicoes*, que é comentado abaixo.

Linha 3: A função *thread_tratadora_de_requisicoes* define e inicializa a estrutura *args*, que recebe a mensagem e o *socket* da conexão com o processo cliente.

```
1 void *thread_tratadora_de_requisicoes(void *__args)
2 {
3       sd_thread_args *args = (sd_thread_args *) __args;
4
5       switch(args->mesg->headr.call)
6       {
7               case 101: {inserir_ED(args); break;}
8               case 103: {localizar_ED(args); break;}
9               case 104: {localizar_ips_ED(args); break;}
10              default:
11                printf("\nSD - DELAULT: Erro no Switch: mm_d.headr.call invalido\n");
12      }
13      close(args->connfd);
14      free(args->mesg);
15      free(args);
16      printf("\nSD - Fim da Thread %d\n",pthread_self());
17      return NULL;
18 }
```

Figura 5.25 - Código da função *thread_tratadora_de_requisicoes*.

Linhas 5 – 12: O operador *switch,* da linguagem C, verifica qual é a operação solicitada pelo processo cliente, verificando o campo *args->mesg->headr.call***,** e desvia a execução do programa para o código correspondente.

Linhas 13 – 18: Depois que o código da operação requisitada pelo processo cliente é executado, a conexão com o processo cliente é fechada, o espaço de memória alocado para a estrutura *args* é liberado e a *thread* tratadora de requisições é encerrada.

Como visto na figura 5.25, o operador *switch* desvia a execução do programa, conforme o identificador da operação. Na realidade, o servidor de diretório oferece basicamente três tipos de serviços: inserir uma nova entrada na Estrutura de Diretório; localizar uma entrada na Estrutura de Diretório e localizar o endereço IP de todos os servidores de arquivos que se cadastraram na estrutura. As funções que correspondem a esses serviços são apresentadas abaixo:

inserir_ED

Essa função é representada pelo identificador de operação 101. Ela geralmente é requisitada pelos servidores de arquivos, logo após a sua inicialização, para inserir na

Estrutura de Diretório os prefixos das entradas de diretório que esses servidores estão disponibilizando e seu número IP.

A figura 5.26 apresenta o código da função *inserir_ED,* que é comentado abaixo.

Linhas 3 – 4: Define a variável *retorno* e cria a estrutura *mesg* que recebe a mensagem de requisição enviada pelo servidor de arquivos.

Linhas 5 – 19: A função *insere_entrada* descrita na seção 5.2.2, implementada no arquivo *Funcoes.h*, é chamada para inserir uma nova entrada na Estrutura de Diretório. Os argumentos passados são o prefixo e o número IP do servidor de arquivos. Se a entrada for inserida com sucesso, a função retorna 1. Se o valor retornado for 0, indica que a nova entrada não foi inserida na Estrutura de Diretório, atualizando a variável *errno*. O resultado retornado pela função é empacotado numa mensagem de resposta, que é enviada para o servidor de arquivos.

Linhas 20 – 21: A mensagem de resposta é enviada ao servidor de arquivos, usando o operador *s_Send.*

```
1 void inserir_ED(sd_thread_args *args)
2 {
3       int retorno;
4       message *mesg = args->mesg;
5       retorno = insere_entrada(&Estrutura, mesg->u_call.steste101.path,
6                                       mesg->u_call.steste101.addr);
7       if(retorno == 1)
8       {
9               printf("\nSD - A entrada %s - %s foi inserida com sucesso.\n",
10                      mesg->u_call.steste101.path, mesg->u_call.steste101.addr);
11      }
12      else
13      {
14              printf("\n SD Erro: A entrada %s - %s não foi inserida.\n",
15                      mesg->u_call.steste101.path, mesg->u_call.steste101.addr);
16              mesg->u_call.m_reply.m_errno = errno;
17      }
18
19      mesg->u_call.m_reply.result = retorno;
20      s_Send(args->connfd, mesg, sizeof(mesg->u_call.m_reply.result)
21              + sizeof(mesg->u_call.m_reply.m_errno)+ sizeof(mesg->headr));
22 }
```

Figura 5.26 – Código da função *inserir_ED.*

localizar_ED

Essa função é representada pelo identificador de operação 103. Ela é requisitada pela interface do sistema do processo cliente, para localizar um prefixo na Estrutura de

Diretório e, conseqüentemente, o IP do servidor de arquivos que está disponibilizando esse prefixo. A figura 5.27 apresenta o código da função *localizar_ED,* que é comentado logo abaixo.

Linhas 3 – 5: São definidas variáveis e cria-se a estrutura *mesg,* que recebe a mensagem de requisição enviada pelo processo cliente.

Linhas 6 – 17: A função *localiza_entrada,* implementada no arquivo *Funcoes.h*, é executada, passando para ela o prefixo do nome de caminho absoluto do arquivo. Através do prefixo, a função busca na Estrutura de Diretório a entrada que contenha esse prefixo. Se o valor retornado pela função for 1, indica que o prefixo foi encontrado e, conseqüentemente, o número IP do servidor de arquivos que disponibiliza esse prefixo. Se o valor retornado for 0, indica que o prefixo não foi encontrado na Estrutura de Diretório, atualizando a variável *errno*. O valor retornado pela função *localiza_entrada* é acrescentado à mensagem de resposta.

Linhas 19 – 21: A mensagem de resposta é enviada ao processo cliente, usando o operador *s_Send.*

```
1 void localizar_ED(sd_thread_args *args)
2 {
3       int retorno;
4       char ende[16];
5       message *mesg = args->mesg;
6       retorno = localiza_entrada(Estrutura, mesg->u_call.steste103.path, ende);
7       if (retorno == 1)
8       {
9               printf("\n Endereço do arquivo %s é: %s \n", mesg->u_call.steste103.path, ende);
10              strcpy(mesg->u_call.m_reply.u_reply.buffer_reply,ende);
11      }
12      else
13      {
14              printf("\n Erro, entrada %s não encontrada.\n", mesg->u_call.steste103.path);
15              mesg->u_call.m_reply.m_errno = errno; // ARQUIVO NAO ENCONTRADO
16      }
17      mesg->u_call.m_reply.result = retorno;
18
19      s_Send(args->connfd, mesg, sizeof(mesg->u_call.m_reply.result)+
20              sizeof(mesg->u_call.m_reply.m_errno)+
21              sizeof(mesg->u_call.m_reply.u_reply.buffer_reply)+sizeof(mesg->headr));
22 }
```

Figura 5.27 - Código da função *localizar_ED*.

localizar_ips_ED

Essa função é representada pelo identificador de operação 104. Ela é requisitada pela interface do sistema, para localizar na Estrutura de Diretório, os IPs de todos os servidores de arquivos do *cluster* que se cadastraram. A figura 5.28 apresenta o código da função *localizar_ips_ED*, o qual é comentado logo abaixo.

Linhas 3 – 5: São definidas variáveis e é criada a estrutura *mesg* que recebe a mensagem de requisição enviada pelo processo cliente.

Linhas 7 – 18: Essa função chama a função *localiza_ips,* que é executada com o objetivo de localizar, na Estrutura de Diretório, os IPs dos servidores de arquivos que estão disponibilizando informações para o espaço de nomes compartilhado. Se o valor retornado pela função *localiza_ips* for 1, indica que foi encontrado pelo menos um número IP, o qual é retornado para a função *localizar_ips_ED*. Em seguida, todos os números IPs encontrados são copiados para a mensagem de resposta. Se o valor retornado for 0, indica que a Estrutura de Diretório está vazia, atualizando o valor da variável *errno* e acrescentando esse valor à mensagem de resposta.

Linhas 20 – 23: A mensagem de resposta é enviada para a interface do sistema do processo cliente, usando o operador *s_Send.*

```
1 void localizar_ips_ED(sd_thread_args *args)
2 {
3       int retorno;
4       char ips[256];
5       message *mesg = args->mesg;
6
7       retorno = localiza_ips(Estrutura, ips);
8       if (retorno == 1)
9       {
10              printf("\nOs ips são: %s \n", ips);
11              strcpy(mesg->u_call.m_reply.u_reply.buffer_reply,ips);
12      }
13      else
14      {
15              printf("\nErro, IPs não encontrados.\n");
16              mesg->u_call.m_reply.m_errno = errno; // IP NAO ENCONTRADO
17      }
18      mesg->u_call.m_reply.result = retorno;
19
20      s_Send(args->connfd, mesg, sizeof(mesg->u_call.m_reply.result)+
21                      sizeof(mesg->u_call.m_reply.m_errno)+
22                      sizeof(mesg->u_call.m_reply.u_reply.buffer_reply)+
23                      sizeof(mesg->header));
24 }
```

Figura 5.28 - Código da função *localizar_ips_ED.*

6 VALIDANDO A DISTRIBUIÇÃO DO SISTEMA DE ARQUIVOS

Este capítulo apresenta na seção 6.1 três cenários para demonstrar a interação entre os processos no ambiente distribuído implementado. Na seção 6.2 são apresentados os testes realizados para validar essa distribuição, e na seção 6.3 são apresentadas as principais características da abordagem distribuída implementada para o sistema de arquivos no ambiente do *cluster* Clux.

6.1 Cenários da Comunicação Cliente/Servidor

Para facilitar a compreensão do modelo arquitetural descrito na seção 5.1 e dos detalhes da implementação apresentados na seção 5.2, são apresentados a seguir três cenários. A numeração das setas nas figuras 6.2, 6.3 e 6.4 corresponde à ordem temporal de execução de cada cenário.

Para exemplificar, na figura 6.1 é apresentado o código, em linguagem C, de um processo cliente.

```
#include <stdio.h>
#include "fcntl_2.h"
#include "cliente.h"

main()
{
        int fp;
        char filename[256] = "/home/dados/Teste.doc ";

        if ((fp = open(filename,O_RDONLY|O_WRONLY|O_CREAT,S_IRWXU))!= -1)
        {
                printf("\nArquivo %s aberto.\n",filename);
                r_close(fp);
        }
        else
        {
                fprintf(stderr, "\nErro ao abrir o arquivo %s.\n",filename);
                perror("cliente");
        }
}
```

Figura 6.1 – Código em linguagem C de um processo cliente.

Na figura acima, o processo cliente usa somente duas chamadas de sistema implementadas pela *libcsa*, que são *open* e *r_close*. A chamada *open* abre o arquivo "*/home/dados/Teste.doc"* retornando o descritor do arquivo, e a chamada *r_close* fecha o arquivo. Nos cenários 1 e 2 é exemplificada a execução da chamada *open*, e no cenário 3 a execução da chamada *r_close*.

6.1.1 Cenário 1: Abrindo um arquivo, com prefixo e IP na *Cache* Local

No primeiro cenário é apresentada a execução da chamada de sistema *open,* onde o prefixo do nome de caminho absoluto do arquivo e o IP se encontram na *Cache* Local do Cliente. A figura 6.2 apresenta esse exemplo.

1. O processo cliente começa a execução do seu código. Quando ele executa a chamada de sistema *open("/home/dados/Teste.doc",O_RDWR|O_CREAT, S_IRWXU)*, é chamada a função *open* correspondente, implementada na interface do sistema.

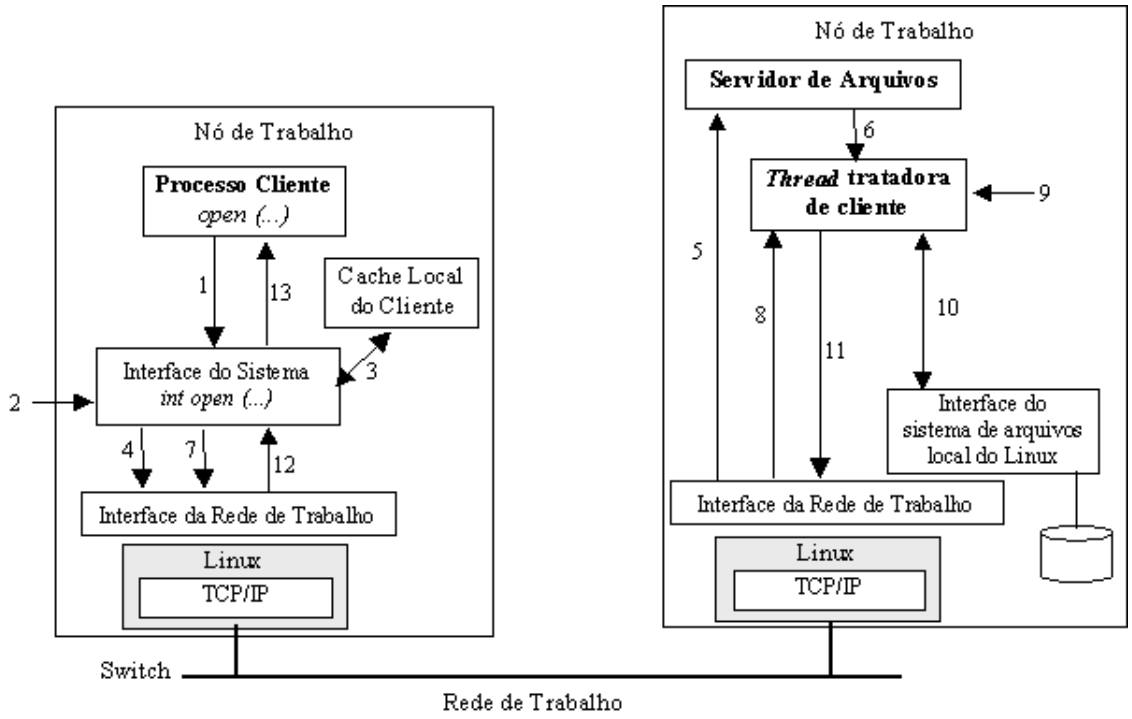


Figura 6.2 – Cenário com prefixo do nome na *Cache* Local do Cliente.

2. A interface do sistema verifica os argumentos da chamada, e em seguida extrai do argumento nome *"/home/dados/Teste.doc"* o prefixo "*/home*".
3. Esse prefixo é usado para verificar se ele já se encontra na *Cache* Local do Cliente. Considerando que o prefixo está na *Cache* Local do Cliente, o IP do servidor de arquivos, que disponibiliza esse prefixo, é lido e retornado.
4. Conhecendo o IP do servidor de arquivos, a interface do sistema estabelece uma conexão com esse servidor, através da interface da rede de trabalho.
5. A interface da rede de trabalho entrega o pedido de conexão ao servidor de arquivos.
6. Quando chega um pedido de conexão, o processo servidor de arquivos cria uma *thread* para tratar da conexão com o cliente. A *thread* tratadora de cliente fica aguardando mensagens de requisição desse cliente.
7. A interface do sistema empacota a identificação da chamada de sistema *open* e seus argumentos numa mensagem de requisição, e a envia ao servidor de arquivos, através da interface da rede de trabalho.
8. A interface da rede de trabalho entrega a mensagem de requisição para a *thread* tratadora de cliente, no servidor de arquivos.
9. Quando chega a mensagem de requisição, a *thread* tratadora de cliente identifica que a chamada de sistema a ser executada é *open,* e desvia a execução do programa para o código associado a essa chamada.
10. A chamada de sistema *open* é executada pela interface do sistema de arquivos local do Linux. Ao término da execução, são realizadas algumas verificações relacionadas ao valor de retorno da chamada.
11. Depois, a mensagem de resposta é montada e enviada para a interface do sistema, no cliente, através da interface da rede de trabalho.
12. A interface da rede de trabalho entrega a mensagem de resposta para a interface do sistema.
13. A função *open*, da interface do sistema, recebe a mensagem, verifica o resultado e repassa esse para o processo cliente. A partir daí, o cliente pode retomar sua execução.

6.1.2 Cenário 2: Abrindo um arquivo, sem prefixo e IP na *Cache* Local

No segundo exemplo é apresentada a execução da chamada de sistema *open,* onde o prefixo do nome de caminho absoluto do arquivo, não se encontra na *Cache* Local do Cliente. A figura 6.3 apresenta esse exemplo.

1. O processo cliente começa a execução do seu código. Quando ele executa a chamada de sistema *open("/home/dados/Teste.doc",O_RDWR|O_CREAT, S_IRWXU)*, é chamada a função *open* correspondente, implementada na interface do sistema.

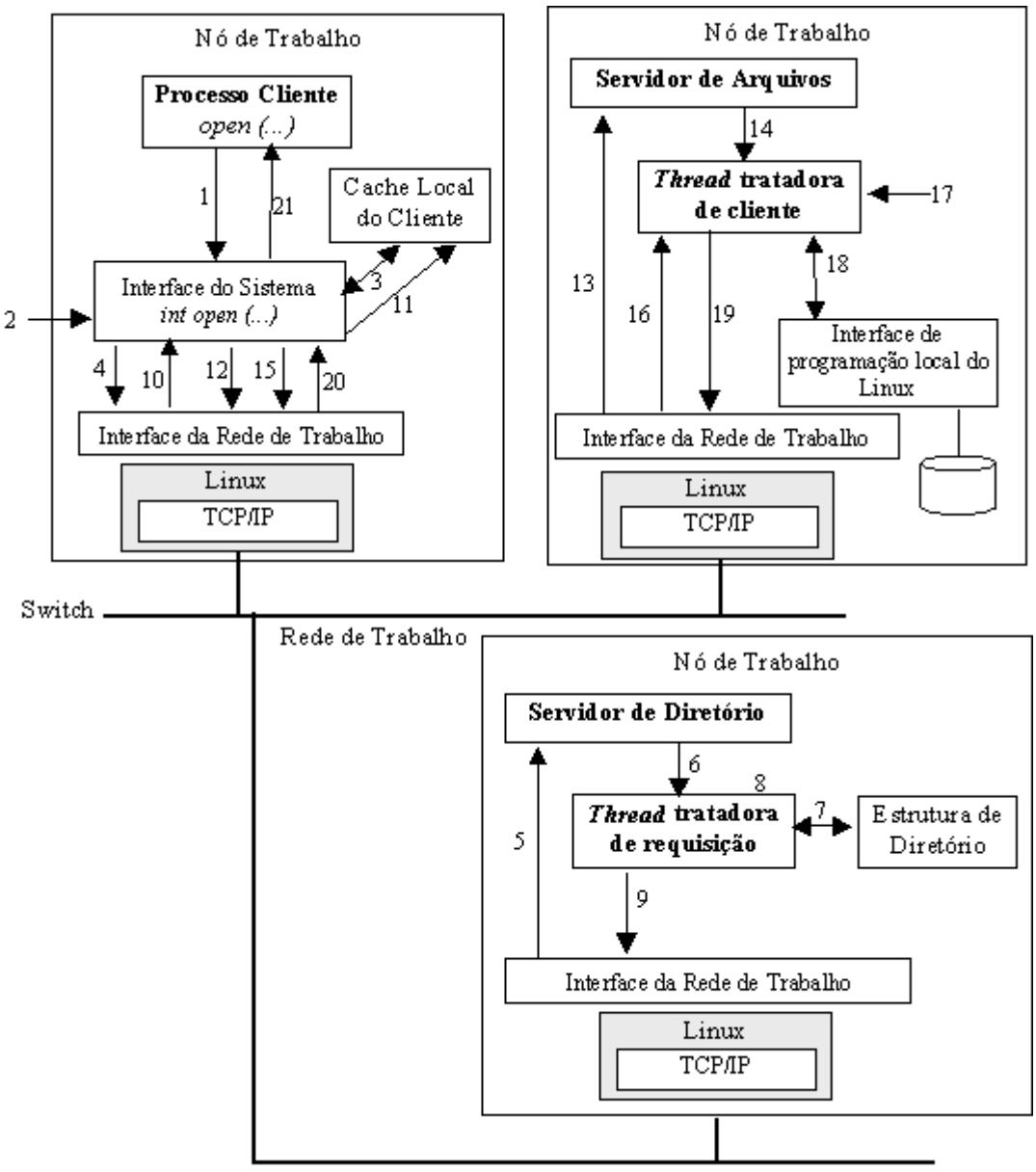


Figura 6.3 – Cenário sem o prefixo do nome na *Cache* Local do Cliente.

2. A interface do sistema verifica os argumentos da chamada, e em seguida extrai do argumento nome *"/home/dados/Teste.doc"* o prefixo "*/home*".
3. É verificado se esse prefixo já se encontra na *Cache* Local do Cliente. Se o prefixo não estiver na *Cache* Local do Cliente, é necessário localizá-lo no servidor de diretório.
4. A interface do sistema estabelece uma conexão com o servidor de diretório e envia para ele uma mensagem de requisição, identificando o serviço e o prefixo procurado, através da interface da rede de trabalho.
5. A interface da rede de trabalho entrega o pedidos de conexão para o servidor de diretório. O servidor de diretório estabelece uma conexão com o cliente e fica aguardando a chegada de mensagens de requisição do processo cliente.
6. Quando chega uma mensagem de requisição, o servidor cria uma *thread* tratadora de requisição, passando para ela o *socket* da conexão e a mensagem de requisição recebida do cliente.
7. A *thread* tratadora de requisição identifica o tipo de serviço requisitado e desvia a execução do programa para o código correspondente, tentando localizar na Estrutura de Diretório o prefixo passado.
8. Considerando, que o prefixo procurado e o IP do servidor de arquivos que disponibiliza esse prefixo, foram encontrados na Estrutura de Diretório, o servidor de diretório empacota o IP numa mensagem de resposta.
9. A mensagem de resposta é enviada para a interface do sistema, no cliente, através da interface da rede de trabalho.
10. A interface da rede de trabalho entrega para a interface do sistema o IP do servidor de arquivos que foi localizado no servidor de diretório.
11. O IP e o prefixo são inseridos na *Cache* Local do Cliente.
12. Conhecendo o IP do servidor de arquivos, a interface do sistema estabelece uma conexão com esse servidor, através da interface da rede de trabalho.
13. A interface da rede de trabalho entrega o pedido de conexão ao servidor de arquivos.
14. Quando chega um pedido de conexão, o processo servidor de arquivos cria uma *thread* para tratar da conexão com o cliente. A *thread* tratadora de cliente fica aguardando mensagens de requisição desse cliente.

15. A interface do sistema empacota a identificação da chamada de sistema *open* e seus argumentos numa mensagem de requisição, e a envia ao servidor de arquivos, através da interface da rede de trabalho.
16. A interface da rede de trabalho entrega a mensagem de requisição para a *thread* tratadora de cliente, no servidor de arquivos.
17. Quando chega uma mensagem de requisição, a *thread* tratadora de cliente identifica a chamada de sistema a ser executada, e desvia a execução do programa para o código associado a essa chamada.
18. A chamada de sistema *open* é executada pela interface do sistema de arquivos local do Linux. Ao término da execução, são realizadas algumas verificações relacionadas ao valor de retorno da chamada.
19. Depois, a mensagem de resposta é montada e enviada para a interface do sistema através da interface da rede de trabalho.
20. A interface da rede de trabalho entrega a mensagem de resposta para a interface do sistema.
21. A função *open* da interface do sistema, recebe a mensagem, verifica o resultado e repassa esse para o processo cliente. A partir daí, o cliente pode retomar sua execução.

6.1.3 Cenário 3: Fechando um arquivo

No terceiro cenário é apresentada a execução da chamada de sistema *r_close*. A figura 6.4 apresenta esse exemplo.

1. Quando o processo cliente executa a chamada de sistema *r_close*, é chamada a função *r_close* correspondente, definida na interface do sistema.
2. A interface do sistema verifica o argumento *fp* da chamada e extrai deste o descritor de arquivo e o *socket* da conexão estabelecida com o servidor de arquivos que disponibiliza esse arquivo.
3. Em seguida, a interface do sistema empacota a identificação da chamada de sistema *r_close* e o descritor de arquivo, numa mensagem de requisição, e a envia ao servidor de arquivos, através da interface da rede de trabalho.
4. A interface da rede de trabalho entrega a mensagem de requisição para a *thread* tratadora de cliente, no servidor de arquivos.

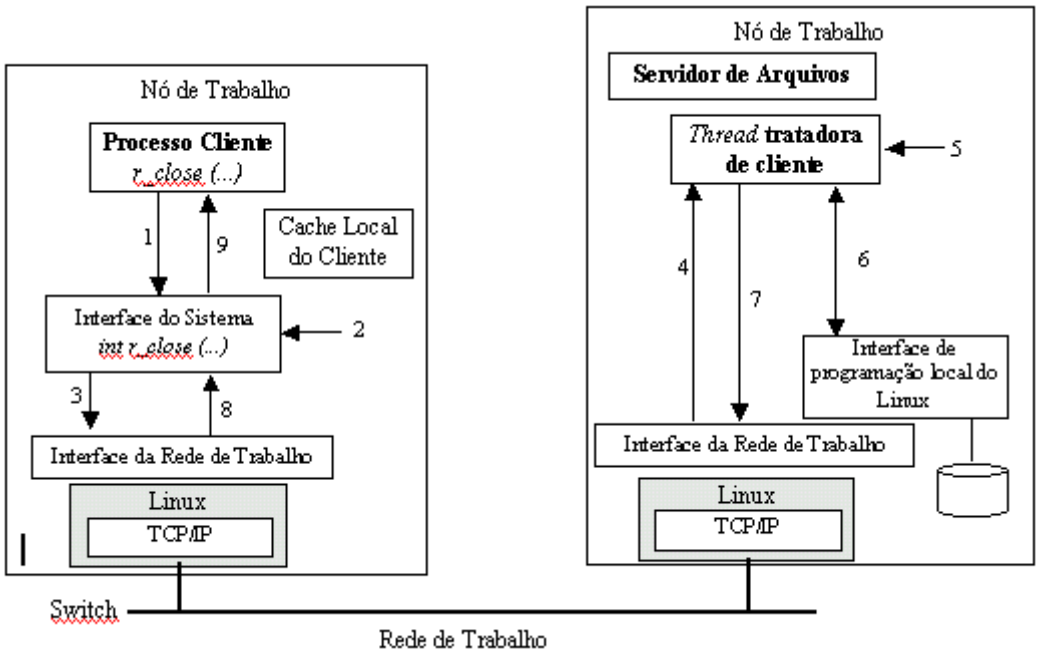


Figura 6.4 – Execução da chamada de sistema *r_close*.

5. Quando chega uma mensagem de requisição, a *thread* tratadora de cliente, verifica que a chamada de sistema a ser executada é *r_close,* e desvia a execução do programa para o código associado a essa chamada.
6. A chamada de sistema *r_close* é executada pela interface do sistema de arquivos local do Linux.
7. Ao término da execução, são realizadas algumas verificações relacionadas ao valor de retorno da chamada. Depois, a mensagem de resposta é montada e enviada para a interface do sistema através da interface da rede de trabalho.
8. A interface da rede de trabalho entrega a mensagem de resposta para a interface do sistema.
9. A função *r_close*, da interface do sistema, recebe a mensagem, verifica o resultado e repassa esse para o processo cliente. A partir daí, o cliente pode retomar sua execução.

6.2 Validação

Para verificar se os objetivos do trabalho foram atingidos, resolveu-se fazer vários testes e analisar os resultados. Através de pesquisas, encontrou-se o *benchmark* Bonnie [BON 96], que implementa certas funções que foram utilizadas para implementar os testes. No Bonnie, os testes basicamente realizam a transferência de blocos de dados entre o espaço do usuário e o disco físico. Cada bloco de um arquivo é lido, em seguida o ponteiro do arquivo é reposicionado para o início do bloco lido, e o bloco é novamente escrito. São analisados a quantidade de *bytes* transferidos por segundo, a utilização da CPU, o tempo do sistema e o tempo de CPU.

Portanto, na seção 6.2.1 são apresentadas as funções do *Bonnie* que foram utilizadas para a implementação dos testes, o algoritmo genérico do teste principal e a apresentação dos diferentes ambientes onde esse teste foi aplicado. Na seção 6.2.2 é feita a comparação dos resultados.

6.2.1 Testes realizados

Os testes foram implementados aproveitando algumas funções do *Benchmark Bonnie*, principalmente aquelas que adquirem e contabilizam os tempos do sistema, como [BON 96]:

- ***timestamp***: Essa função é responsável por ler o tempo atual do sistema.
- ***get_delta_t***: Essa função lê o tempo atual do sistema e em seguida, subtrai esse da última leitura de tempo realizada pela função *timestamp*, para descobrir o tempo que decorreu desde a execução da função *timestamp* até a execução da função *get_delta_t*.

Entre os operadores implementados pela *libcsa*, os mais requisitados pelos processos clientes são: *open, read, lseek, write* e *r_close*. Devido a isso, foi implementado um teste que utiliza essas cinco chamadas.

O algoritmo genérico desse teste, inicialmente, abre um arquivo usando a chamada de sistema *open*. Em seguida, é iniciado um laço *for* que realizará 500 vezes as seguintes operações: leitura de um bloco de 1024 bytes do arquivo (*read*)*;* reposicionamento do ponteiro do arquivo para o início do último bloco lido (*lseek);* e a

escrita do bloco de 1024 bytes recém lido (*write*). Depois do laço *for* terminar, o arquivo é fechado com a chamada *r_close*.

Na realização dos testes, analisou-se o tempo de resposta, ou seja, o tempo decorrido desde o instante que um processo cliente executa um determinado número de chamadas de sistema, até o instante que ele receber as respostas.

Considerando o algoritmo acima, para medir o tempo de resposta, no início do laço *for* é executada a função *timestamp.* Em seguida são executas as chamadas de sistema *read, lseek* e *write.* Os passos da execução de uma chamada de sistema, que possui o identificador único (descritor do arquivo + *socket*), são apresentados no cenário 3 da seção 6.1.3. Ao final do laço *for,* quando as três chamadas foram executadas, executa-se a função *get_delta_t,* que calcula o tempo que decorreu desde a execução da função *timestamp* até a execução da função *get_delta_t*. Para cada iteração do laço *for* o tempo de resposta é contabilizado e armazenado num vetor. Quando o laço *for* termina e o arquivo é fechado, os tempos de resposta armazenados no vetor são usados para calcular o tempo médio de resposta.

Abaixo são apresentados os testes realizados, onde o mesmo programa de teste foi aplicado para diferentes ambientes.

Teste 1

O ambiente utilizado no primeiro teste é apresentado na figura 6.5, o qual possui um nó de trabalho rodando o servidor de arquivos centralizado e dois nós de trabalho rodando vários processos clientes. Foram executados ao mesmo tempo 6 processos clientes, três em cada nó de trabalho. O resultado obtido com esse teste é representado por T1 no gráfico 6.1.

Figura 6.5 – Ambiente 1.

Teste 2

No segundo teste, o ambiente utilizado é composto de um nó de trabalho para rodar o servidor de arquivos, um nó de trabalho para rodar o servidor de diretório e dois nós de trabalho para rodar os processos clientes. Esse ambiente é apresentado na figura 6.6. Os processos servidores e clientes fazem parte da abordagem distribuída implementada para o sistema de arquivos neste trabalho, porém, este teste utiliza somente um nó de trabalho com o serviço de arquivos.

Foram executados, de maneira concorrente, 6 processos clientes, três no primeiro nó de trabalho e os outros três no segundo nó de trabalho. O resultado obtido nesse teste é representado por T2 no gráfico 6.1.

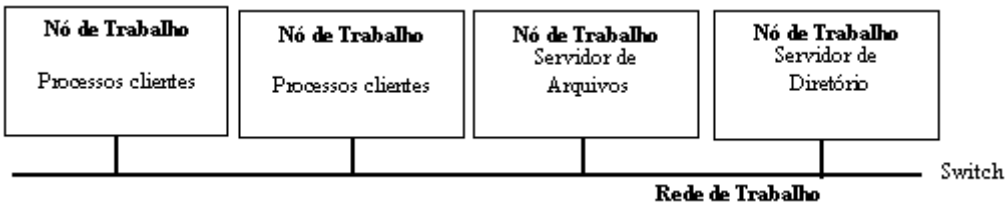


Figura 6.6 – Ambiente 2.

Teste 3

O terceiro teste foi realizado no ambiente apresentado na figura 6.7, onde o serviço de arquivos foi disponibilizado em dois nós de trabalho.

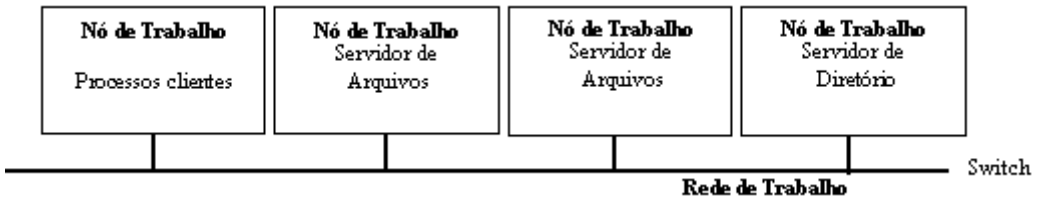


Figura 6.7 – Ambiente 3.

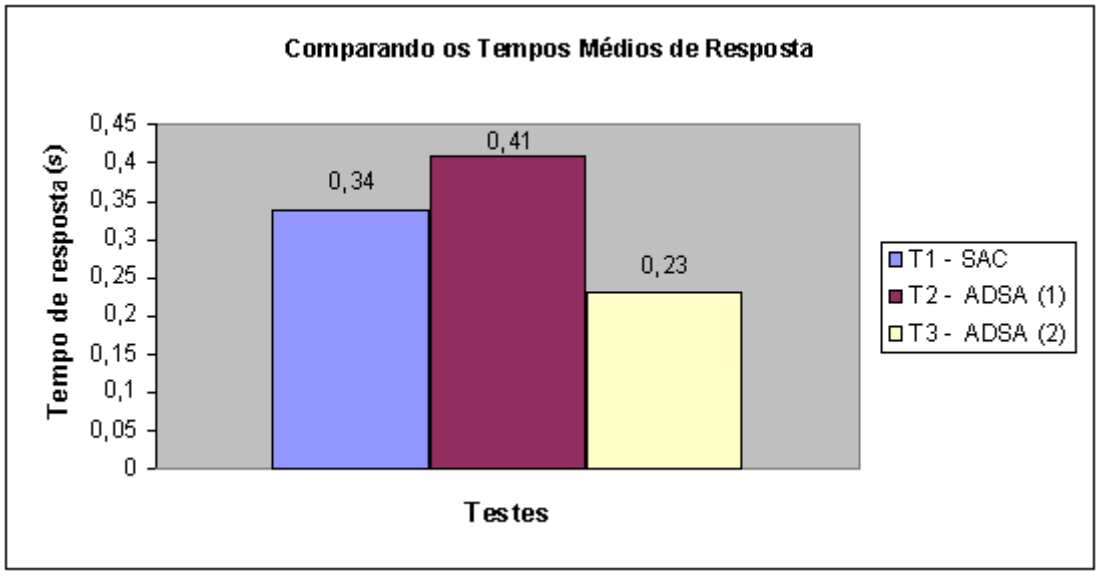
Esse teste tem como objetivo verificar se o tempo médio de resposta aumenta ou diminui, quando o serviço de arquivos é disponibilizado em dois nós de trabalho. Através dos tempos produzidos, calculou-se o tempo médio de resposta, o qual é

representado no gráfico 6.1 por T3. Depois de realizar o teste, observou-se que o tempo médio de resposta diminuiu com a distribuição do serviço de arquivos.

6.2.2 Comparando os resultados

Para compreender melhor os resultados dos testes realizados, o gráfico 6.1 faz uma comparação entre os tempos médios de resposta destes. Na legenda do gráfico, T1 representa o teste 1, T2 o teste 2 e T3 o teste 3.

Gráfico 6.1 – Comparando os tempos médios de resposta dos testes.



Para os testes T1 e T2, conclui-se que num *cluster,* com somente um nó de trabalho disponibilizando o servidor de arquivos, implementado para a abordagem distribuída, o tempo médio de resposta será maior, do que num *cluster* com um servidor de arquivos centralizado. Isso ocorre, porque na implementação da abordagem distribuída para o sistema de arquivos, são usadas várias estruturas para manter e gerenciar determinadas informações, como a *Cache* Local do Cliente, a Estrutura de Diretório e a Tabela de Conexões, as quais não existem no servidor de arquivos centralizado.

Comparando T1 com T3, observa-se que quando o serviço de arquivos é disponibilizado em dois nós de trabalho, o tempo médio de resposta para o cliente diminui. Para confirmar os resultados do terceiro teste, disponibilizou-se o servidor de

arquivos, implementado para a abordagem distribuída, em três nós de trabalho, o servidor de diretório em um único nó de trabalho e os seis processos clientes, executando ao mesmo tempo, em outro nó de trabalho. Observou-se, que o tempo médio de resposta para cada cliente diminuiu ainda mais.

Portanto, pode-se dizer que a distribuição do serviço de arquivos num ambiente de *cluster* ajuda a diminuir o tempo de resposta para o processo cliente, aumentando a disponibilidade do serviço e conseqüentemente o desempenho do sistema.

6.3 Características do Sistema Implementado

Comparando-se os sistemas de arquivos distribuídos estudados no capítulo 3 e apresentados de forma sucinta na Tabela 3.1, com a abordagem distribuída do sistema de arquivos implementada, pode-se dizer que a abordagem distribuída para o *cluster* Clux apresenta as seguintes características:

- **Servidor sem estado:** O servidor de arquivos é sem estado, não mantendo nenhum tipo de informação sobre quais clientes estão acessando quais arquivos.
- ***Cache*:** Não é realizada a *cache* de arquivos no cliente, mas usa-se a memória principal, no cliente, para fazer *cache* dos prefixos dos nomes de caminhos requisitados pelos processos clientes e os IPs dos servidores que os disponibilizam.
- **Replicação:** Não é implementado nenhum tipo de replicação de dados ou serviços.
- **Consistência:** Como o servidor de arquivos é sem estado, fica difícil garantir a consistência dos arquivos. A única forma de consistência oferecida é a semântica UNIX.
- **Escalabilidade:** Não implementa replicação de dados nem *cache* dos arquivos no cliente. A implementação distribuída do sistema de arquivos para o Clux permite aumentar o número de servidores de arquivos disponíveis no ambiente do Clux, porém, implementa somente um servidor

de diretório. Assim, pode-se dizer que a escalabilidade do ambiente fica limitada.

- **Desempenho:** O desempenho pode ser aumentado, disponibilizando uma maior quantidade de servidores de arquivos no ambiente. Também usa de *cache* no cliente para guardar os prefixos e IPs, evitando novas conexões com o servidor de diretório.
- **Persistência dos dados quando ocorrem falhas:** Não é implementado nenhum mecanismo de tolerância a falhas. É usada a semântica UNIX.
- **Resolução de nomes:** Usa-se as informações da *Cache* Local do Cliente ou as informações da Estrutura de Diretório para traduzir o prefixo do nome de caminho de um arquivo no IP do servidor de arquivos que o disponibiliza.
- **Espaço de nomes compartilhado:** Todos os clientes tem a mesma visão do espaço de nomes compartilhado.
- **Segurança:** Não implementa nenhum tipo de segurança específico. Usa somente a segurança oferecida pelo Linux, ou seja, o controle de acesso baseado em bits de proteção.
- **Nós clientes sem disco:** Permite nós clientes sem disco. Os processos clientes simplesmente requisitam a realização de operações sobre os arquivos no servidor de arquivos remoto.

7 CONCLUSÃO

Neste capítulo, na seção 7.1 são apresentadas as contribuições deste trabalho, e na seção 7.2 são descritos os projetos futuros.

7.1 Contribuições

Este trabalho apresentou o projeto e a implementação da distribuição do sistema de arquivos para um ambiente de *cluster*. Os passos necessários para sua implementação foram uma revisão bibliográfica na área de sistemas distribuídos, dando uma ênfase maior para multicomputadores heterogêneos, também conhecidos como *clusters*, e um aprofundamento no estudo dos sistemas de arquivos distribuídos. Em seguida, estudou-se o ambiente do *cluster* Clux, principalmente, o servidor de arquivos centralizado, implementado com processos regulares do Linux, baseado no modelo cliente/servidor, que usa uma interface de comunicação baseada em *sockets* TCP/IP.

A partir desse estudo e de alguns trabalhos de implementação, começou-se a definir como o sistema de arquivos seria distribuído no ambiente do *cluster.* Em seguida, iniciou-se um trabalho de implementação, onde os produtos finais foram um servidor de arquivos e um servidor de diretório *multithreaded*, além da interface do sistema adaptada para um ambiente distribuído.

A implementação do servidor de diretório tinha como objetivo, identificar para o processo cliente o servidor de arquivos que disponibilizava determinado arquivo para o espaço de nomes compartilhado do *cluster*, permitindo que um processo cliente manipulasse o arquivo, sem conhecer sua localização física, atingindo a meta de transparência de localização para os clientes.

O servidor de arquivos foi re-implementado, substituindo os processos regulares do Linux por *threads,* tornando-se um servidor de arquivos *multithreaded*.

Para diminuir a quantidade de mensagens trocadas entre os processos clientes e o servidor de diretório, criou-se uma estrutura de *Cache* Local, na memória principal do cliente, onde são mantidos os prefixos disponibilizados pelos servidores de arquivos e

seus IPs. Os processos clientes interagem com os servidores, através da interface da rede de trabalho, implementada com *sockets* TCP/IP.

Para a distribuição do sistema de arquivos no *cluster*, disponibilizou-se o servidor de arquivo em dois nós de trabalho e o servidor de diretório num único nó de trabalho. Em seguida, realizou-se testes em ambientes diferentes, observando-se que o tempo médio de resposta do sistema diminuía quando o serviço de arquivos era disponibilizado em mais de um nó de trabalho.

Portanto, como a distribuição do serviço de arquivos no *cluster* foi implementada, permitindo aos processos clientes acessar os arquivos do espaço de nomes compartilhado de forma transparente, sem conhecer sua localização física; diminuindo a quantidade de requisições enviadas ao processo servidor de diretório, com o uso da estrutura de *cache* local no processo cliente; aumentando a disponibilidade do serviço de arquivos e conseqüentemente o desempenho do sistema; conclui-se que o objetivo do trabalho foi atingido.

7.2 Perspectivas Futuras

Quando se iniciou o desenvolvimento da abordagem distribuída do sistema de arquivos, objetivava-se utilizar como interface de comunicação, a interface da rede de trabalho e a interface da rede de controle, ambas desenvolvidas especialmente para o *cluster* Clux. Porém, as interfaces apresentavam muitos problemas e não atendiam as expectativas. Devido a isso, sugere-se, como proposta de trabalho futuro, que essas interfaces sejam re-implementadas num outro trabalho de dissertação.

Durante a pesquisa e o desenvolvimento da abordagem distribuída do sistema de arquivos, pôde-se notar que existe uma grande quantidade de serviços que poderiam ser acrescentados ao sistema para melhorá-lo. Entre os serviços propostos teria-se:

- A implementação de um serviço de replicação de dados, que ajudaria a aumentar a disponibilidade das informações e a escalabilidade do sistema.
- Permitir que o primeiro nível do espaço de nomes compartilhado possa ser gerenciado e modificado pelo servidor de diretório, além de atualizar as informações da *cache* local de cada cliente.

- Implementação de mecanismos de tolerância a falhas, que garantissem, que quando um servidor falhasse, outro servidor pudesse assumir o serviço, de maneira transparente para o cliente.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[ANT 99] Antônio, M. *Sistemas Operacionais Distribuídos.* 1999. Acesso em: 10 de outubro de 2002. Disponível em: <http://orbita.starmedia.com/~brodowski/sem2.htm>.

[BAC 86] Bach, M.J, *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[BOG 02] Bogo, Madianita. *Interface da Rede de Controle do Cluster Clux.* Dissertação de Mestrado, CPGCC / UFSC, 2002.

[BON 96] Bonnie, *Benchmark*. *Benchmark Bonnie.* Revisado em 1996. Acesso em 16 de setembro de 2004. Disponível em: <http://www.textuality.com/bonnie/>.

[BUD 02] Budag, C.H. *Implementação do núcleo do sistema operacional distribuído ACrux.* Dissertação de Mestrado. CPGCC / UFSC, 2002.

[BUE 02] Bueno, A. D; *Introdução ao Processamento Paralelo e ao Uso de Clusters de Workstations em Sistemas GNU/LINUX.* LMPT EMC UFSC Acesso em: 10 de outubro de 2003. Disponível em: <http://www.rau-tu.unicamp.br/nou-rau/softwarelivre/document>.

[CAD 93] Card, R.; Ts'o, T.; Tweedie, S. *Design and Implementation of the Second Extended Filesystem.* Primeiro Simpósio Internacional Holandês de Linux,1993. Acesso em: 10 de outubro de 2003. Disponível em: <http://e2fsprogs.sourceforge.net/ext2intro.html>

[CAR 00] Carvalho, R. P. *Sistemas de Arquivos Distribuídos.*, Trabalho, IME / USP, 2000. Acesso em: 12 de setembro de 2003. Disponível em: <http://www.ime.usp.br/~carvalho>.

[CLU 00] *Cluster Computing White Paper.* Dezembro de 2000. Editora Mark Baker, Universidade de Portsmouth, UK. Acesso em: 15 de outubro de 2003. Disponível em: <http://www.fisica.uson.mx/carlos/LinuxClusters/cluster-computing2000.pdf > .

[COR 99] Corso, T. B. *Crux: Ambiente Multicomputador Reconfigurável por Demanda*, Tese de Doutorado, CPGEE / UFSC, 1999.

[COU 01] Coulouris, G., Dollimore, J., Kindberg, T. *Distributed Systems. Concepts and Design*. 3 ed., Pearson Education, 2001.

[DAI 02] Daines, B. *Cluster Opinion: The Ethernet Opportunity for Linux Clusters.* Linux Network, 2002. IEEE Task Force on Cluster Computing. Acesso em: 10 de outubro de 2003. Disponível em: <http://www.clustercomputing.org/index.jsp?page=/content/tfcc-5-2-daines.shtml>.

[FAU 95] Fausto, L. F. *PYXIS: Um Servidor de Nomes para Ambientes Distribuídos.* CGCC, UFSC, Florianópolis, 1995.

[FER 01] Ferreto, T. C, et al. *CPAD-PUCRS/HP: GNU/LINUX como plataforma para pesquisa em alto desempenho*., Porto Alegre, RS, 2001. Acesso em: 10 de outubro de 2003. Disponível em: <http://www.ulbra.tche.br/wsl2001/anais.pdf>.

[FLY 72] Flynn, M. J. *Some Computer Organizations and Their Effectiveness*. IEEE Trans. On Computers. Vol. C-21. Pp. 948-960. Setembro, 1972.

[FRO 94] Fröhlich, A. A. M. *Pyxis: Um Sistema de Arquivos Distribuído*. Dissertação de Mestrado, CPGCC / UFSC, 1994.

[HOC 03] Hochstetler, S.; Beringer, B. *Linux Clustering with CSM and GPFS.* IBM Global Services, IBM Brasil. Acesso em: 15 de outubro de 2003. Disponível em: < http://www.redbooks.ibm.com/redbooks/SG246601.html>.

[KON 96] Fabio Kon. *Distributed File Systems Past, Present and Future.* Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1996. Acesso em: 10 de outubro de 2003. Disponível em: <http://www.ime.usp.br/~kon/>.

[MPI 03] MPI (*Message Passing Interface*). Acesso em: 10 de outubro de 2003. Disponível em: <http://www.netlib.org/mpi/readme>.

[MYR 03] *Myrinet.* Acesso em: 10 de outubro de 2003. Disponível em: <http://www.myricom.com/myrinet/>.

[PLE 01] Plentz, Patrícia Della Méa. *Um servidor de arquivos para um cluster de computador.* Dissertação de Mestrado, CPGCC / UFSC, 2001.

[PVF 00] Carns, P. H.; Ligon, W. B., *PVFS: A Parallel Virtual File System for Linux Clusters*. Technical report, Parallel Architecture Research Laboratory, Clemson University, 2000. Acesso em: 10 de novembro de 2003. Disponível em: <http://www.parl.clemson.edu/pvfs/el2000/extreme2000.html#sec:pvfs_design>.

[PVM 03] PVM (*Parallel Virtual Machine*). Acesso em: 10 de outubro de 2003. Disponível em: <http://www.netlib.org/pvm3>.

[REC 02] Rech, Luciana de Oliveira. *Interface da Rede de Trabalho do Cluster Clux*. Dissertação de Mestrado, CPGCC / UFSC, 2002.

[RIB 99] Ribeiro, C. C., Rodriguez, N. R. *Otimização e Processamento Paralelo de Alto Desempenho.* Revista PUC Ciências, dezembro de 1999. Acesso em: 10 de outubro de 2003. Disponível em: http://www-di.inf.puc-rio.br/~celso/publicacoes_bottom.htm.

[RIC 00] Ricarte, Ivan Luiz Marques. *Programação de Sistemas: Uma Introdução.* Acesso em: 9 de agosto de 2003. Disponível em <http://www.dca.fee.unicamp.br/~ricarte>, Campinas, fevereiro de 2000.

[SOA 95] Soares, L. F. *Redes de Computadores: das LANs, MANs e WANs às redes ATM*. Rio de Janeiro: Campus, 1995.

[TAN 01] Tanenbaum, A. S., Steen, M. van. *Distributed Systems. Principles and Paradigms*, Prentice Hall, Publicado em setembro de 2001.

[TAN 03] Tanenbaum, A. S. *Sistemas Operacionais Modernos,* Prentice Hall, 2003.

[TAN 92] Tanenbaum, A. S. *Sistemas Operacionais Modernos,* Prentice Hall, 1992.

[TAN 97] Tanenbaum, A. S. *Redes de Computadores,* 3ª edição, Rio de Janeiro, Campus, 1997.

[VAZ 03] Vaz, T. B, *Clusters Beowulf.* Arquitetura de computadores. Instituto de Matemática, DCC, UFBA. Novembro de 2003. Acesso em: 23 de novembro de 2003. Disponível em: <http://coisa.im.ufba.br/~tiago/arq/apresentacao_beowulf_1.0.pdf>.