

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Patricia Della Mía Plentz

**UM SERVIDOR DE ARQUIVOS PARA
UM CLUSTER DE COMPUTADORES**

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

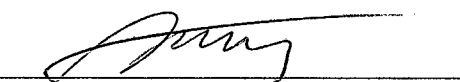
Prof. Dr. Thadeu Botteri Corso

Florianópolis, dezembro de 2001.


UM SERVIDOR DE ARQUIVOS PARA UM CLUSTER DE COMPUTADORES

Patricia Della Méa Plentz

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.

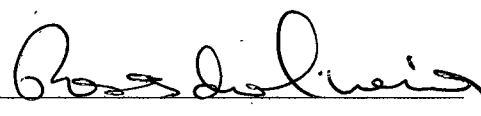


Prof. Fernando A. O. Gauthier, Dr.
(Coordenador)




Prof. Thadeu Botteri Corso, Dr.
(Orientador)

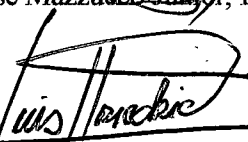
Banca Examinadora



Prof. Rômulo Silva de Oliveira, Dr.



Prof. José Mazzucco Junior, Dr.



Prof. Luis Fernando Friedrich, Dr.

Sumário

RESUMO	IX
ABSTRACT	X
1. INTRODUÇÃO	1
1.1 MOTIVAÇÕES.....	1
1.2 OBJETIVOS.....	1
1.3 ORGANIZAÇÃO DO TEXTO.....	2
2. SISTEMAS OPERACIONAIS	3
2.1 ASPECTOS GERAIS.....	3
2.1.1 Chamadas de sistema.....	4
2.1.2 Processos.....	5
2.1.3 Arquivos.....	5
2.1.4 Programas de Sistema.....	6
2.2 MODOS DE OPERAÇÃO.....	6
2.3 ESTRUTURAS.....	7
2.3.1 Sistemas Monolíticos.....	7
2.3.2 Sistemas em Camadas.....	8
2.3.3 Sistemas Cliente-Servidor.....	10
2.4 SISTEMA DE ARQUIVOS.....	10
2.4.1 Implementação de Arquivos.....	11
2.4.2 Estrutura Interna do Arquivo.....	11
2.4.3 Diretórios.....	12
2.5 UNIX.....	12
2.5.1 Aspectos Gerais.....	12
2.5.2 Processos.....	14
2.5.3 Sistema de Arquivos.....	15
2.5.4 Chamadas de Sistema.....	16
2.6 LINUX.....	19
2.6.1 Introdução.....	19

2.6.2 Sistema de Arquivos.....	19
3. PROTOCOLOS DE COMUNICAÇÃO.....	21
3.1 REDES DE COMPUTADORES.....	21
3.1.1 Redes Locais.....	21
3.1.2 Redes Metropolitanas.....	24
3.1.3 Redes Geograficamente Distribuídas.....	25
3.2 ARQUITETURA TCP/IP.....	26
3.3 TCP (<i>TRANSMISSION CONTROL PROTOCOL</i>).....	27
3.3.2 Estabelecendo uma conexão.....	29
3.3.3 Encerrando uma conexão.....	30
3.4 UDP (<i>USER DATAGRAM PROTOCOL</i>).....	30
3.5 SOCKETS.....	32
3.6 SERVIDORES CONCORRENTES.....	36
4. SISTEMAS COMPUTACIONAIS COM MEMÓRIA DISTRIBUÍDA	37
4.1 MULTICOMPUTADORES.....	37
4.1.1 Classificação.....	37
4.1.2 Redes de Interconexão.....	38
4.1.3 Ambiente Multicomputador Crux.....	42
4.2 CLUSTER DE COMPUTADORES.....	43
4.2.1 Visão Geral.....	43
4.2.2 Sistema Operacional em Clusters.....	44
4.2.3 Mecanismos de Comunicação em Clusters.....	45
4.2.4 Projetos de Clusters.....	49
4.2.5 Tabela Comparativa.....	53
5. SERVIDOR DE ARQUIVOS	54
5.1 CLUSTER ALVO.....	54
5.2 MODELO OPERACIONAL.....	55
5.2.1 Interface do Sistema.....	56
5.2.2 Interface da Rede de Trabalho.....	57
5.3 DETALHES DE IMPLEMENTAÇÃO.....	59
5.3.1 Comunicação Cliente-Servidor.....	60

5.3.2 <i>Estrutura da mensagem</i>	60
5.3.3 <i>Processos Clientes</i>	63
5.3.4 <i>Servidor de Arquivos</i>	64
5.4 EXEMPLO	65
6. CONCLUSÃO	74
6.1 CONTRIBUIÇÕES	74
6.2 PERSPECTIVAS FUTURAS	75
7. REFERÊNCIAS BIBLIOGRÁFICAS	76

Lista de Figuras

Figura 2.1 – Chamada de Sistema.	4
Figura 2.3 – Sistema monolítico.	8
Figura 2.4 – Estrutura do sistema operacional THE.	9
Figura 2.5 – Modelo cliente-servidor.	10
Figura 2.7 – Chamadas mais comuns que tratam da gerência de processos no Unix. ...	17
Figura 2.8 – Chamadas mais comuns que tratam da manipulação de arquivos e diretórios no Unix.	18
Figura 3.1 – Topologia em Estrela.	22
Figura 3.2 – Topologia em Anel.	23
Figura 3.3 – Topologia em Barra.	24
Figura 3.4 – Arquitetura da rede metropolitana.	25
Figura 3.5 – Organização da arquitetura TCP/IP.	26
Figura 3.6 – O segmento TCP.	28
Figura 3.7 – Handshake de Três Vias.	29
Figura 3.8 – Handshake de Três Vias utilizado para encerrar conexões.	30
Figura 3.10 – Servidor concorrente com <i>sockets</i> TCP/IP.	36
Figura 4.1 – Grelha.	39
Figura 4.2 – Hipercubo.	39
Figura 4.3 – Torus.	40
Figura 4.4 – Barramento.	40
Figura 4.5 – Rede Ômega 8 x 8.	41
Figura 4.6 – Crossbar 4 x 4.	41
Figura 4.7 – Arquitetura do multicomputador Crux.	42
Figura 4.8 – Estrutura genérica de um cluster.	44
Figura 5.1 – Arquitetura genérica do cluster alvo.	54
Figura 5.2 – Servidor de Arquivos.	56
Figura 5.3 – Operador <i>Send</i>	57
Figura 5.4 – Operador <i>Receive</i>	58
Figura 5.6 – Formato da mensagem.	61

Figura 5.7 – Estrutura da mensagem definida na linguagem C.	61
Figura 5.8 – Mensagem correspondente à chamada de sistema <i>open</i>	62
Figura 5.9 – Chamada de sistema <i>open</i> definida na linguagem C.	62
Figura 5.10 – Mensagem de retorno definida na linguagem C.	63
Figura 5.11 – Algoritmo genérico das funções da biblioteca <i>libcsa</i>	64
Figura 5.12 – Algoritmo genérico do servidor de arquivos.	65
Figura 5.13 – Código do processo cliente.	66
Figura 5.15 – Código da função <i>inicializa_cliente</i>	68
Figura 5.17 – Código da função <i>inicializa_servidor</i>	72

Lista de Tabelas

Tabela 4.1 – Principais características dos projetos.	53
--	----

Resumo

O objetivo deste trabalho é o de propor um servidor de arquivos para um cluster de computadores derivado do multicomputador Crux [COR99]. Este cluster oferece um ambiente para a execução de programas paralelos organizados como redes de processos comunicantes. A arquitetura do cluster é composta de um determinado número de nós de trabalho e um nó de controle. Os nós de trabalho são interligados por uma rede de interconexão dinâmica reconfigurável conforme a demanda dos programas paralelos. Esse servidor é implementado com processos regulares do sistema operacional Linux, executando em um único nó de trabalho. Processos clientes, executando em outros nós de trabalho, comunicam-se com o processo servidor de arquivos pela troca de mensagens através da rede de interconexão dinâmica.

Abstract

The objective of this work is to propose a file server for a cluster of computers derived from the multicomputer Crux [COR99]. This cluster offers an environment for the execution of parallel programs organized as networks of communicating processes. The cluster architecture is formed by a definite number of work nodes and one control node. Work nodes are linked through a dynamic interconnection network reconfigured by the demand of parallel programs. This server is implemented as regular processes of the operating system Linux, executing in one work node. Clients processes, executing in others work nodes, exchange messages with the file server through the network of dynamic interconnection.

1. Introdução

Este capítulo objetiva identificar as motivações e objetivos gerais deste trabalho, bem como apresentar a organização do texto. No tópico 1.1 são apresentadas as motivações, no tópico 1.2 são descritos os objetivos e no tópico 1.3 é apresentada a organização do texto.

1.1 Motivações

Clusters de computadores apresentam-se, cada vez mais, como uma alternativa atraente de ambiente para processamento paralelo. Um cluster é formado por um conjunto de computadores pessoais ou estações de trabalho interligados por uma rede, montados com componentes comerciais. Tecnologias de redes de interconexão como Fast Ethernet [COM98] e Myrinet [MYR01], assim como pacotes de software para ambientes paralelos e distribuídos como PVM [PVM01] e MPI [RIB01] colaboram na exploração do potencial de paralelismo dos clusters de computadores.

A montagem de um cluster de computadores derivado do multicomputador Crux [COR99] é objeto de projetos correntes no ambiente do Laboratório de Computação Paralela e Distribuída (LaCPaD) do Curso de Pós-Graduação em Ciência da Computação (CPGCC) da Universidade Federal de Santa Catarina (UFSC).

A operação do cluster depende da implementação de mecanismos de comunicação específicos do multicomputador Crux que estão sendo adaptados para o cluster e pressupõe a existência de um servidor de arquivos centralizado.

1.2 Objetivos

O objetivo principal deste trabalho se refere ao projeto e implementação de um servidor de arquivos para um cluster derivado do multicomputador Crux, implementado com processos regulares do Linux, seguindo o modelo cliente-servidor.

De um lado, um processo cliente faz uma solicitação de serviço ao sistema de arquivos; a solicitação é colocada em uma mensagem e enviada ao processo servidor. De outro lado, o processo servidor recebe mensagens com requisições características de um sistema de arquivos como *open*, *read*, *write* e *close*; faz o processamento dessa requisição e retorna o resultado ao processo cliente que fez a solicitação.

1.3 Organização do Texto

O capítulo 2 apresenta os principais conceitos relacionados aos sistemas operacionais, e descreve os sistemas operacionais Unix e Linux. O capítulo 3 mostra os aspectos mais relevantes sobre os protocolos de comunicação para redes de computadores, enfatizando o protocolo TCP/IP. O capítulo 4 introduz os multicomputadores e os clusters de computadores. O capítulo 5 apresenta o servidor de arquivos, objeto deste trabalho, para o cluster alvo derivado do multicomputador Crux, descrevendo a estrutura criada e os detalhes da implementação. O capítulo 6 mostra as contribuições deste trabalho e projetos futuros.

2. Sistemas Operacionais

Este capítulo introduz os principais conceitos relacionados aos sistemas operacionais. No tópico 2.1 são apresentados os aspectos gerais dos sistemas operacionais, como os conceitos de processos, arquivos, chamadas de sistema e programas de sistema. No tópico 2.2 são descritos os dois modos de operação normalmente implementados pelos processadores. No tópico 2.3 são apresentadas as principais estruturas de sistemas operacionais. No tópico 2.4 são mostrados os principais aspectos relacionados aos sistemas de arquivos. Nos tópicos 2.5 e 2.6 são descritos os sistemas operacionais Unix e Linux, respectivamente.

2.1 Aspectos Gerais

O sistema operacional de um computador é a parte de software que estende os recursos de hardware da máquina, tornando a utilização do equipamento mais fácil, mais eficiente e mais confiável [OLI00]. Ele é formado por um grupo de procedimentos que prestam serviços aos usuários do sistema e suas aplicações, assim como a outras rotinas do próprio sistema. As principais funções desse conjunto de rotinas são:

- Tratamento de interrupções;
- Criação e eliminação de processos;
- Escalonamento e controle dos processos;
- Sincronização e comunicação entre processos;
- Gerência de memória;
- Gerência de entrada/saída;
- Gerência de arquivos;
- Contabilização e segurança.

A interface entre o sistema operacional e os programas de usuário é definida pelo conjunto de instruções estendidas fornecidas pelo sistema. Tais instruções, conhecidas como chamadas de sistema, manipulam os objetos gerenciados pelo sistema

operacional. A seguir são introduzidos os conceitos de chamadas de sistema, processos, arquivos e programas de sistema.

2.1.1 Chamadas de sistema

Cada sistema operacional tem seu próprio conjunto de chamadas de sistema. Os programas de usuário solicitam serviços do sistema operacional através da execução de chamadas de sistema. Para cada um desses serviços existe uma chamada de sistema relacionada [MAC97]. A figura 2.1 ilustra este contexto.

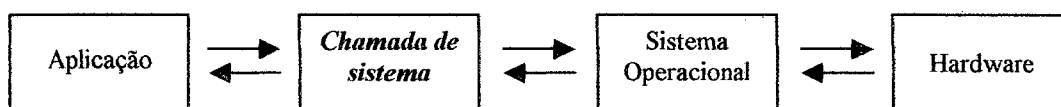


Fig. 2.1 – Chamada de Sistema.

O processamento das solicitações de serviços dos programas de usuário é realizado de acordo com os argumentos da chamada de sistema, e a resposta é retornada à aplicação.

No decorrer da execução de um programa de usuário podem ocorrer eventos que desviam a execução do processador para rotinas que tratam esses eventos. Esse desvio, conhecido como mecanismo de interrupção, faz com que o programa de usuário em execução seja interrompido e uma rotina específica, chamada tratador de interrupção, seja executada. O tratador de interrupção executa as ações necessárias para tratar esse evento. Após a execução desta rotina, o processamento retorna, do ponto em que parou, ao programa de usuário. Para que isso seja possível, o processador ao identificar uma interrupção, armazena na pilha de execução, os registradores internos do processador.

2.1.2 Processos

Um processo é basicamente um programa em execução, incluindo os valores correntes de todos os registradores do hardware e das variáveis manipuladas por ele no curso de sua execução.

Para implementar o modelo de processo, o sistema operacional deve manter uma tabela, chamada tabela de processos, com uma entrada por processo. Cada entrada inclui informações sobre o estado do processo, sobre sua prioridade, sobre a memória alocada, sobre os valores de seu ponteiro de instruções e ponteiro de pilha, sobre o estado de seus arquivos abertos, entre outras.

2.1.3 Arquivos

Arquivos são constituídos de informações que se relacionam entre si, podendo representar programas ou dados. Os arquivos são administrados pelo sistema operacional para tornar fácil o acesso dos usuários ao seu conteúdo. A parte do sistema responsável por essa gerência é denominada *sistema de arquivos*.

Uma das principais funções do sistema operacional é esconder as particularidades dos dispositivos de entrada/saída, e apresentar ao programador um modelo abstrato com arquivos independentes dos dispositivos onde estão implementados.

Os arquivos são identificados através de um nome, o qual é formado por um conjunto de caracteres de tamanho variável. As regras variam de um sistema de arquivos para outro. Alguns, por exemplo, limitam o tamanho máximo do nome do arquivo e fazem a diferenciação entre caracteres alfabéticos maiúsculos e minúsculos no nome do arquivo. Em alguns sistemas operacionais, a identificação do nome do arquivo é dividido em duas partes separadas por um ponto. A parte seguinte ao ponto é denominada extensão do arquivo e indica alguma característica do arquivo.

O sistema operacional suporta diversas operações sobre arquivos. As principais operações são [OLI00]:

- Criação do arquivo;
- Destruição do arquivo;

- Leitura do conteúdo;
- Alteração do conteúdo;
- Escrita de novos dados;
- Execução do programa contido no arquivo;
- Alteração do nome do arquivo.

2.1.4 Programas de Sistema

Os programas de sistema realizam tarefas básicas para a utilização do sistema. São executados fora do núcleo do sistema operacional e utilizam as mesmas chamadas de sistema disponíveis aos programas de usuário [OLI00].

O interpretador de comandos é o mais importante programa de sistema. Ele é ativado pelo sistema operacional sempre que um usuário inicia sua sessão de trabalho. Sua tarefa é receber comandos do usuário e processá-los. Normalmente, a execução do comando exige uma ou mais chamadas de sistema.

Os utilitários para manipulação de arquivos são um exemplo de programa de sistema. Eles listam, imprimem, copiam e alteram o nome de arquivos, entre outras funções. A utilização desses programas de sistema também é comum para a obtenção de informações a respeito do sistema, tal como data, hora ou quais usuários estão utilizando o computador em um dado momento.

2.2 Modos de Operação

A utilização indevida de algumas instruções causa sérios problemas à integridade do sistema. Estas instruções não podem, portanto, ser colocadas diretamente à disposição das aplicações. Operações de entrada/saída são exemplos deste tipo de instrução.

As *instruções não-privilegiadas* são as que não oferecem perigo ao sistema, enquanto que aquelas que tem o poder de comprometer o sistema são chamadas de *instruções privilegiadas*.

O mecanismo de *modos de operação* é implementado pelo processador para que aplicações possam executar instruções privilegiadas. É comum o uso de dois modos de operação implementados pelo processador: *modo usuário* e *modo supervisor*. No modo usuário apenas as instruções não-privilegiadas podem ser executadas; no modo supervisor todas as instruções do processador podem ser executadas.

Uma forma de controlar o acesso às instruções privilegiadas é permitir que somente o sistema operacional tenha acesso à elas. Assim, o sistema operacional executa com o processador no modo supervisor e os programas de usuário executam em modo usuário. O sistema operacional, antes de entregar o processador aos processos de usuário, muda para o modo usuário. Assim, os processos de usuários executam apenas as instruções não-privilegiadas.

2.3 Estruturas

Existem diversas maneiras de estruturar um sistema operacional, sendo as mais conhecidas a estrutura monolítica, a estrutura em camadas e a estrutura cliente-servidor. Estas estruturas são apresentadas a seguir.

2.3.1 Sistemas Monolíticos

Em sistemas monolíticos o sistema operacional é organizado como um conjunto de procedimentos, cada um dos quais podendo interagir com os demais sempre que necessário. Neste tipo de sistema não existe uma estruturação visível, como mostra a figura 2.3

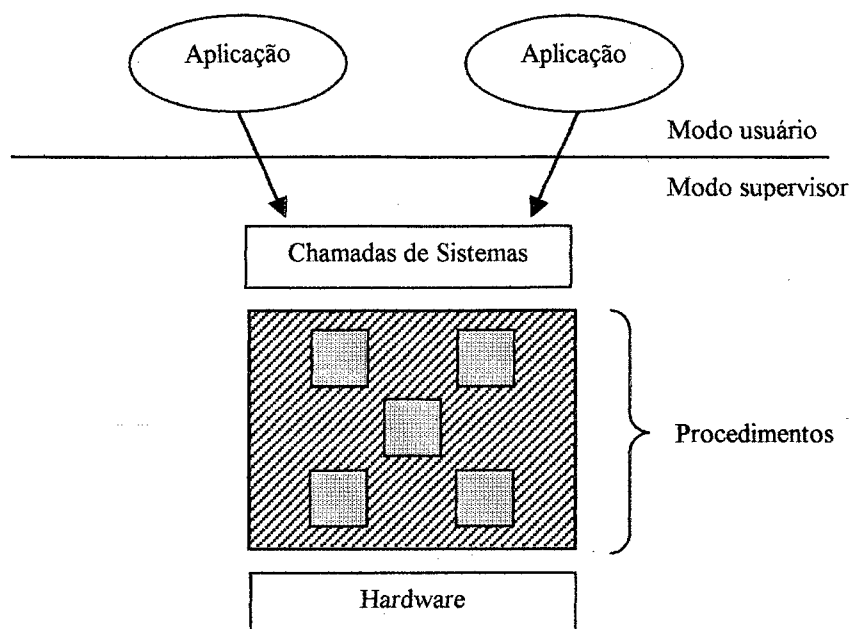


Fig. 2.3 – Sistema monolítico.

Cada procedimento do sistema deve ter uma interface bem definida em termos de argumentos e resultados [TAN92]. Quando este modelo for usado na construção do programa objeto relativo ao sistema operacional, os procedimentos individuais são compilados e ligados, formando um único arquivo objeto.

Restrições relacionadas à visibilidade dos procedimentos não são implementadas. Desta maneira, cada procedimento é visível a todos os outros. Nesse tipo de sistema, as chamadas de sistemas são requisitadas através da colocação de argumentos em lugares bem determinados, como registradores ou pilhas, seguindo-se uma instrução para alterar o modo de acesso do modo usuário para o modo supervisor. Analisando os argumentos o sistema operacional define qual chamada de sistema deve ser executada. Para sua execução, a chamada de sistema pode acionar os procedimentos do sistema operacional necessários ao serviço solicitado. Após a conclusão da execução da chamada de sistema, o controle retorna ao programa do usuário.

2.3.2 Sistemas em Camadas

Nos sistema em camadas o sistema operacional é dividido em camadas de software sobrepostas. Essas camadas possuem funções que podem ser utilizados por

outras camadas. Funções de uma camada podem fazer referência à funções da camada imediatamente inferior ou de camadas inferiores.

O primeiro sistema a fazer uso desta abordagem foi o sistema THE [TAN92] que utilizava seis camadas. A figura 2.4 apresenta a estrutura do sistema operacional THE. A camada 0 (camada inferior) permite a multiprogramação do processador, realizando a sua alocação e chaveamento entre processos quando ocorrem interrupções, ou quando o quantum do tempo expira. A camada 1 realiza a alocação de espaços para processos na memória e em um disco de 512 K endereços, que armazenava partes de processos para as quais não havia espaço na memória principal. A camada 2 realiza as comunicações entre cada processo e o console de operação. A camada 3 realiza a administração dos dispositivos de entrada/saída, movendo informações de/para tais dispositivos. A camada 4 armazena os programas de usuário e na camada 5 (camada superior) está o operador do sistema.

5	Operador
4	Programas de usuário
3	Gerência dos dispositivos de entrada/saída
2	Comunicação processo-operador
1	Gerência da memória
0	Alocação do processador

Fig. 2.4 – Estrutura do sistema operacional THE.

Posteriormente, os sistemas MULTICS [TAN92] e VMS [MAC97] também implementaram o conceito de camadas.

A estruturação em camadas apresenta algumas vantagens, como por exemplo, a possibilidade de isolar as funções do sistema operacional, o que facilita a sua alteração e depuração, e também, protege as camadas mais internas na medida que cria uma hierarquia de níveis [MAC97].

2.3.3 Sistemas Cliente-Servidor

Neste modelo, a maioria das funções tradicionais dos sistemas operacionais são implementadas como processos de usuário. O núcleo do sistema fica responsável pela gerência de processos e pela comunicação entre processos clientes e processos servidores. Um processo cliente é aquele que solicita algum serviço ao sistema e o processo servidor é aquele que executa este serviço. Um processo cliente solicita um serviço enviando uma requisição ao processo servidor, o qual realiza o trabalho solicitado e envia a resposta ao processo cliente. A figura 2.5 ilustra este modelo.

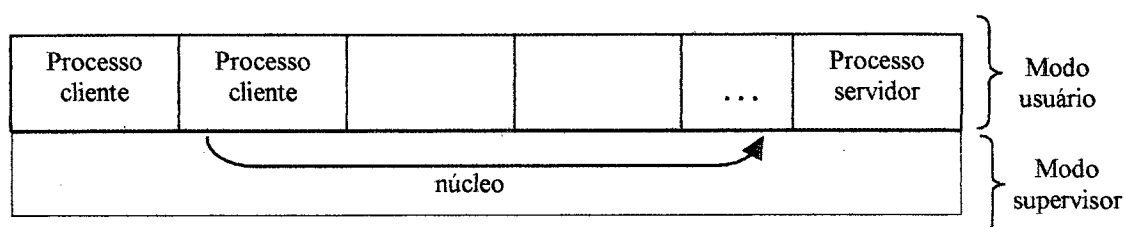


Fig. 2.5 – Modelo cliente-servidor.

Dividindo o sistema operacional em vários servidores, cada um tratando um aspecto do sistema, tais como o sistema de arquivos e o gerente de memória, é possível tornar cada parte do sistema menor e mais fácil de implementar.

O modelo cliente-servidor possibilita que os servidores executem em modo usuário e o núcleo do sistema operacional execute em modo supervisor. Assim, se ocorrer um erro em um servidor, o sistema não ficará totalmente comprometido. Apenas o servidor que sofreu o problema ficará parado. Esta estrutura também permite melhor manutenção porque as funções do sistema operacional podem ser isoladas em processos servidores pequenos e dedicados a serviços específicos.

2.4 Sistema de Arquivos

Um sistema de arquivos é o conjunto de métodos e estruturas de dados que um sistema operacional utiliza para administrar arquivos em um disco magnético.

2.4.1 Implementação de Arquivos

A cada arquivo no sistema operacional está associado um descritor de arquivo. Este descritor é um registro que mantém informações sobre o arquivo, tais como: nome, tamanho, local no disco onde o conteúdo do arquivo foi colocado, data e hora do último acesso, data e hora da última alteração, identificação do usuário que criou o arquivo e a lista de usuários que podem acessar o arquivo e respectivos direitos de acesso. O descritor de arquivos fica armazenado em disco, usualmente na mesma partição onde se encontra o conteúdo do arquivo [OLI00].

Na maioria dos sistemas operacionais, um processo de usuário que queira utilizar um arquivo deve primeiramente solicitar a abertura deste arquivo. Isto acontece através das chamadas de sistema e serve para o sistema de arquivos localizar o arquivo no disco e carregar o descritor deste arquivo para a memória.

O sistema de arquivos mantém então na memória uma cópia dos descritores dos arquivos abertos. Quando o arquivo é fechado, o sistema de arquivos atualiza o descritor do disco, caso ele esteja desatualizado em relação à cópia na memória.

2.4.2 Estrutura Interna do Arquivo

A estrutura interna de um arquivo corresponde ao modo como os seus dados estão armazenados. Várias formas podem ser usadas para estruturar um arquivo.

Uma forma de estruturar os arquivos é através de uma seqüência não-estruturada de bytes, na qual o sistema operacional vê o arquivo como uma seqüência de bytes, sem que nenhuma estrutura lógica seja estabelecida para os dados. A aplicação fica responsável pelo controle de acesso aos arquivos. A vantagem deste método está na flexibilidade para criar diferentes estruturas de dados.

Outra forma de estruturação é definir os arquivos como uma seqüência de registros de tamanho fixo, onde cada registro possui uma estrutura interna característica. Desta maneira, uma operação de leitura retorna um registro e uma operação de escrita grava um registro.

Uma terceira forma de organização coloca os arquivos como uma árvore de registros. Os registros tem tamanhos variados e cada registro contém um campo chave

em uma posição fixa. A ordenação da árvore acontece pelo campo chave, permitindo uma busca rápida de determinada chave [TAN92].

2.4.3 Diretórios

Diretórios são conjuntos de referências à arquivos [OLI00]. Um diretório é uma estrutura de dados que contém entradas associadas aos arquivos como seu nome, localização física e organização. Quando um arquivo é aberto, o sistema operacional busca, a partir do nome, os demais atributos do arquivo (armazenados diretamente no diretório ou em uma estrutura de dados apontada pelo diretório), colocando tais informações em uma tabela na memória. Todas as referências subseqüentes ao arquivo usam as informações da memória principal.

2.5 Unix

Este tópico apresenta o sistema operacional Unix. Primeiramente, no subtópico 2.5.1 são introduzidos os aspectos gerais, no subtópico 2.5.2 são mostrados os conceitos fundamentais, e por fim, no subtópico 2.5.3 são apresentadas as chamadas de sistema.

2.5.1 Aspectos Gerais

O Unix é um sistema operacional multitarefa e multi-usuário interativo de tempo compartilhado. Ele gerencia o hardware do computador alocando recursos, escalonando tarefas, processando requisições de usuários e executando funções administrativas e de manutenção para o sistema.

Atualmente, o sistema operacional Unix é um importante padrão que influenciou o projeto de muitos sistemas operacionais modernos e apresenta os seguintes recursos e capacidades:

- Centenas de programas utilitários para executar uma grande variedade de funções como criação, edição, e manipulação de arquivos e textos,

processamento de comandos e tarefas, comunicação com outros usuários, manutenção do sistema e desenvolvimento de programas;

- O interpretador de comandos (shell), que funciona como uma interface com o usuário, é uma ferramenta flexível que habilita usuários a executar o seu trabalho ao mesmo tempo que provê uma estrutura que separa e protege usuários e seus ambientes uns dos outros e do sistema operacional;
- Sistema de arquivos e sistema de entrada/saída simplificado, onde cada arquivo, comando, programa e dispositivo de entrada/saída é tratado pelo sistema operacional como um arquivo que contém cadeias de caracteres;
- Unix foi projetado para ser portátil de forma que ele possa ser implementado facilmente em várias plataformas computacionais.

Um sistema Unix pode ser visualizado como uma espécie de pirâmide. No nível mais baixo está o hardware, envolvendo o processador, memória, dispositivos de entrada/saída, além de outros dispositivos. No nível seguinte está o sistema operacional Unix, controlando o hardware e fornecendo as chamadas de sistema para que todos os programas tenham acesso aos serviços do sistema. No próximo nível da pirâmide encontra-se a biblioteca de procedimentos-padrão, com um procedimento para cada chamada de sistema. Então, para executar um comando de leitura, um programa na linguagem de programação C chama o procedimento *read* da biblioteca. No nível seguinte estão os programas utilitários-padrão. Nele se incluem o processador de comandos (shell), os compiladores, os editores, os programas para processamento de texto, e os utilitários para manipulação de arquivos. Alguns são especificados na recomendação POSIX 1003.2, e alguns são diferentes para as várias versões do Unix. No último nível estão os usuários que utilizam os programas citados anteriormente. A figura 2.6 apresenta estes níveis citados acima.

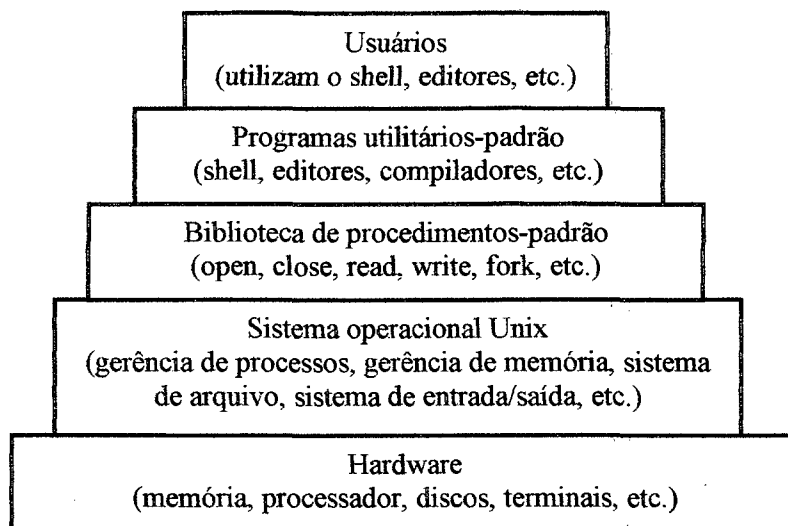


Fig. 2.6 – Níveis de um sistema Unix.

Os comandos do Unix são processados pelo shell, que faz parte dos programas utilitários-padrão. O shell é responsável por interpretar os comandos do usuário, convertendo-os em chamadas do sistema operacional.

2.5.2 Processos

No Unix os processos são elementos ativos no sistema porque alteram o seu estado durante a execução de um programa [OLI00]. Cada processo executa um único programa, e tem uma única linha de controle, isto é, tem apenas um ponteiro de instruções.

Cada processo criado possui um número associado a ele chamado de *process id* (pid). Este número distingue o processo de todos os outros processos criados e que ainda não terminaram sua execução. Assim, cada processo possui um pid único. Quando um processo cria outro, ele é chamado de processo pai e o processo criado é chamado processo filho. No Unix, um processo pai cria um processo filho fazendo cópia de si mesmo. Com exceção do pid, estes processos (pai e filho) são idênticos. Para executar um novo programa, o processo criado sobrepõe o código e os dados do programa herdado do processo pai pelo código e os dados do novo programa e então inicia sua execução. Um processo pai pode ter vários processos filho e um processo filho tem um único processo pai.

Um processo pai pode suspender sua própria execução. Por exemplo, o shell normalmente executa um programa criando um processo filho e esperando que este filho termine sua execução para liberar o caracter de prompt para o usuário.

O único processo no Unix que não tem pai é o processo 0, o qual é criado durante a inicialização do sistema. O ancestral de todos os outros processos no sistema é o processo 1. Ele é criado pelo processo 0 e chamado de *init*. Os processos são escalonados de acordo com uma prioridade, que é definida e ajustada dinamicamente pelo núcleo do Unix.

Processos conhecidos como *daemon* são processos especiais, que respondem por tarefas especiais no sistema, como por exemplo, gerência de filas de impressão e escalonamento de processos *batch*.

2.5.3 Sistema de Arquivos

Os conceitos básicos no Unix relacionados ao sistema de arquivos são superbloco, nó-i, bloco de dados, bloco de diretórios e bloco de indireção. O superbloco contém as informações sobre o sistema de arquivos como um todo, como por exemplo, seu tamanho. Um nó-i contém as informações sobre um determinado arquivo, exceto seu nome. O nome está armazenado no diretório, junto com o número do nó-i. Uma entrada de diretório é formada pelo nome e pelo número do nó-i que representa o arquivo. O nó-i contém o número de diversos blocos de dados usados para armazenar as informações do arquivo. Há espaço somente para uns poucos blocos de dados no nó-i, e, caso um número maior seja necessário, mais espaço para ponteiros será alocado dinamicamente. Estes blocos alocados dinamicamente são blocos de indireção que, como o nome indica, contém endereços para outros blocos de dados.

O Unix suporta diversas operações sobre arquivos, tais como: criação, destruição, leitura do conteúdo, alteração do conteúdo, troca do nome do arquivo, execução do programa contido no arquivo.

No Unix, os arquivos podem ser agrupados em diretórios, por conveniência de seus usuários. Os diretórios são armazenados como se fossem arquivos, e, de uma forma geral, podem ser tratados como arquivos [TAN92]. O sistema de arquivos assume uma forma hierárquica na medida em que os diretórios podem conter subdiretórios.

Uma forma de se especificar nomes de arquivos em Unix é através do chamado *caminho absoluto*, no qual é informado como chegar ao arquivo a partir do diretório raiz. Porém, nomes absolutos são muito grandes e por vezes inconveniente. Assim, o *caminho relativo* é uma outra maneira de especificar nomes de arquivos. O Unix permite que seus usuários designem o diretório no qual estão trabalhando como diretório de trabalho ou diretório atual. Todos os nomes de caminhos são interpretados em relação ao diretório atual. Por exemplo, se o diretório atual for *usr/ast*, então, o arquivo cujo caminho absoluto é *usr/ast/abcd* pode ser referenciado apenas por *abcd*.

2.5.4 Chamadas de Sistema

Os programas solicitam serviços ao sistema operacional através das chamadas de sistema. Elas são semelhantes à subrotinas. Entretanto, enquanto as subrotinas são procedimentos normais de um programa, as chamadas de sistema ativam o sistema operacional. Através de argumentos, o programa informa exatamente o que necessita. O retorno da chamada de sistema faz com que a execução do programa seja retornada à partir da instrução que segue à chamada [OLI00].

Quando ocorre algum erro na execução das chamadas de sistema, o Unix utiliza a variável global *errno* para indicar o tipo de erro que ocorreu. Esta variável recebe então um valor positivo que informa o tipo de erro e a chamada de sistema retorna em geral o valor -1 . No arquivo *errno.h* estão as constantes que são associadas a esses valores positivos.

A seguir, é apresentada uma seleção das chamadas de sistema mais representativas do POSIX (*Portable Operating System Interface*).

Chamadas de Sistema para gerência de processos

Gerência de Processo	Descrição
<code>pid = fork()</code>	Cria um processo filho
<code>s = waitpid (pid, &status, opts)</code>	Espera até que um filho termine
<code>exit (status)</code>	Termina a execução e retorna um código de estado
<code>s = execve (name, argv, envp)</code>	Substitui a imagem de memória de um processo

Fig. 2.7 - Chamadas mais comuns que tratam da gerência de processos no Unix.

O `fork` é a chamada de sistema que cria um novo processo no Unix exatamente igual ao processo original, incluindo descritores de arquivos, registradores e tudo mais.

A chamada `waitpid` faz com que um processo pai espere o término da execução de um processo filho. Essa chamada possui três argumentos, sendo que o primeiro argumento permite que o processo pai espere por um processo filho específico. No caso desse argumento ser `-1`, o processo pai espera pelo primeiro dos filhos que terminar. O segundo argumento informa o endereço de uma variável que receberá o código do estado de saída do filho e o último argumento informa se o processo pai retorna sua execução ou fica bloqueado, caso nenhum filho tenha ainda terminado sua execução.

A chamada de sistema `exit` é invocada por cada um dos processos ao término de sua execução e possui apenas um argumento, que é o estado de saída desse processo.

A chamada de sistema `execve` substitui a imagem do processo atual pelo novo arquivo de programa e este novo programa normalmente inicia na função `main`. `execve` tem três argumentos: o nome do arquivo a ser executado, um ponteiro para o vetor de argumentos e um ponteiro para o vetor de ambiente.

Chamadas de Sistema para arquivos e diretórios

Arquivos e Diretórios	Descrição
fd = open (file, flags, mode)	Abre um arquivo para leitura e/ou escrita
s = close (fd)	Fecha um arquivo aberto
n = read (fd, buffer, nbytes)	Lê dados de um arquivo e os coloca em um <i>buffer</i>
n = write (fd, buffer, nbytes)	Escreve os dados de um buffer em um arquivo
pos = lseek (fd, offset, whence)	Move o ponteiro do arquivo para qualquer posição
s = mkdir (name, mode)	Cria um novo diretório
s = rmdir (name)	Remove um diretório vazio

Fig. 2.8 – Chamadas mais comuns que tratam da manipulação de arquivos e diretórios no Unix.

A chamada de sistema `open` abre (mas também pode criar) um arquivo. Os argumentos necessários para esta chamada de sistema são o nome do arquivo, `flags` (nesse argumento várias opções podem ser especificadas) e modo de acesso.

A chamada de sistema `close` fecha um arquivo. Como argumento necessita apenas do descritor de arquivo.

A chamada de sistema `read` lê um bloco de bytes de um arquivo. Possui três argumentos: descritor de arquivo, endereço origem do bloco e tamanho do bloco (em bytes).

A chamada de sistema `write` escreve um bloco de bytes em um arquivo. O segundo argumento informa o endereço do bloco destino e os demais argumentos são idênticos aos da chamada de sistema `read`.

A chamada de sistema `lseek` permite que programas acessem qualquer parte do arquivo. O primeiro argumento especifica o descritor de arquivo, o segundo é a posição do arquivo e o terceiro argumento informa se a posição é relativa ao início do arquivo, à posição corrente ou ao final do arquivo. O retorno desta chamada é a posição absoluta no arquivo após a mudança do ponteiro.

As chamadas de sistema `mkdir` e `rmdir` são usadas para criar e destruir diretórios, respectivamente.

2.6 Linux

Neste t3pico s3o abordados os principais conceitos relacionados ao sistema operacional Linux. No subt3pico 2.6.1 3 feita uma breve introdu33o e no subt3pico 2.6.2 s3o apresentados os conceitos que envolvem o sistema de arquivos do Linux.

2.6.1 Introdu33o

O Linux 3 um sistema operacional pertencente 3 fam3lia Unix e segue as especifica33es POSIX. Ele foi projetado inicialmente para computadores PCs baseados em processadores 386, mas hoje roda em v3rios tipos de m3quinas diferentes como Sparcs da Sun, Amiga e Power PCs. Mas, ao contr3rio do Unix, o Linux possui o c3digo fonte aberto. Tal caracter3stica permite que qualquer pessoa veja como o sistema funciona, corrija alguma problema ou fa3a alguma sugest3o sobre sua melhoria, e esse 3 um dos motivos de seu r3pido crescimento, do aumento da compatibilidade de perif3ricos e de sua estabilidade.

Existem variados tipos de distribu33es do Linux, sendo que por distribu33es entende-se o conjunto formado pelo sistema operacional b3sico e pelo conjunto de aplicativos e suas configura33es. Exemplos de distribu33es s3o Red Hat, Debian, Mandrake e Slackware.

Juntamente com a possibilidade de escolha entre as diversas distribu33es, existem tamb3m as op33es de interfaces gr3ficas, tornando esse sistema operacional ainda mais atrativo. Exemplos de interfaces gr3ficas s3o o KDE e GNOME.

2.6.2 Sistema de Arquivos

A organiza33o do sistema de arquivos do Linux 3 em forma de uma 3rvore hierarquizada. Assim, cada arquivo, diret3rio e dispositivo de entrada/s3ida 3 representado por um n3 da 3rvore.

O Linux suporta diferentes sistemas de arquivos, possibilitando a exist3ncia simult3nea, em disco, dele e outros sistemas operacionais. Neste sentido, o conceito de parti33es e pontos de montagem s3o importantes. Um disco f3sico pode ser dividido em v3rios discos l3gicos, e para cada disco l3gico pode ser associado um sistema de

arquivos. Esses discos lógicos são chamados de partições. Os pontos de montagem são os diretórios. No Linux, o diretório “/” é definido como o ponto de montagem inicial e a partir dele é possível criar novos diretórios, bem como definir outros sistemas de arquivos [OLI00].

O sistema de arquivos do Linux é o *ext2* (*Second Extended File System*), sendo que o bloco é a estrutura básica desse sistema. O bloco pode conter informações referentes ao sistema de arquivos como um todo, ou conter dados.

3. Protocolos de Comunicação

O objetivo principal deste capítulo é introduzir os protocolos de comunicação para redes de computadores. Inicialmente, no tópico 3.1 é apresentado o conceito de redes de computadores e sua classificação. No tópico 3.2 é feita uma breve introdução à arquitetura TCP/IP. No tópico 3.3 é mostrado o protocolo TCP (*Transmission Control Protocol*) e a seguir, no tópico 3.4, é apresentado o protocolo UDP (*User Datagram Protocol*). O tópico 3.5 descreve o conceito de *sockets* para a arquitetura TCP/IP e o tópico 3.6 introduz o conceito de servidores concorrentes.

3.1 Redes de Computadores

As redes de computadores são constituídas de módulos processadores, interligados por um sistema de comunicação, sendo capazes de trocar informações e compartilhar recursos. Um módulo processador pode ser qualquer dispositivo que se comunique por troca de mensagens através do sistema de comunicação, como por exemplo, um terminal, uma impressora ou um computador.

O sistema de comunicação é formado por um arranjo topológico que interliga os módulos processadores por meio de enlaces físicos e de um conjunto de regras para organizar a comunicação (protocolos).

A seguir é apresentada a classificação das redes de computadores que se subdividem em redes locais, redes metropolitanas e redes geograficamente distribuídas [SOA95].

3.1.1 Redes Locais

Redes Locais (LANs – *Local Area Networks*) são redes privadas contidas normalmente em um prédio, que tem alguns quilômetros de extensão. As redes locais foram definidas e utilizadas inicialmente nos ambientes de institutos de pesquisa e universidades. Elas surgiram para viabilizar a troca e o compartilhamento de

informações e dispositivos periféricos preservando a independência das várias estações de processamento. Atualmente são amplamente usadas para interligar computadores pessoais e estações de trabalho em escritórios, universidades e instalações industriais.

As redes locais tradicionais operam em velocidades que podem variar de 10 a 100 Mbps, tem um baixo retardo e cometem pouquíssimos erros [TAN97]. São apresentadas a seguir as topologias mais utilizadas nestas redes.

Topologia em Estrela

Na topologia em estrela um nó central interliga os demais nós. Todas as mensagens devem passar pelo nó central, o qual age como um centro de controle da rede (figura 3.1).

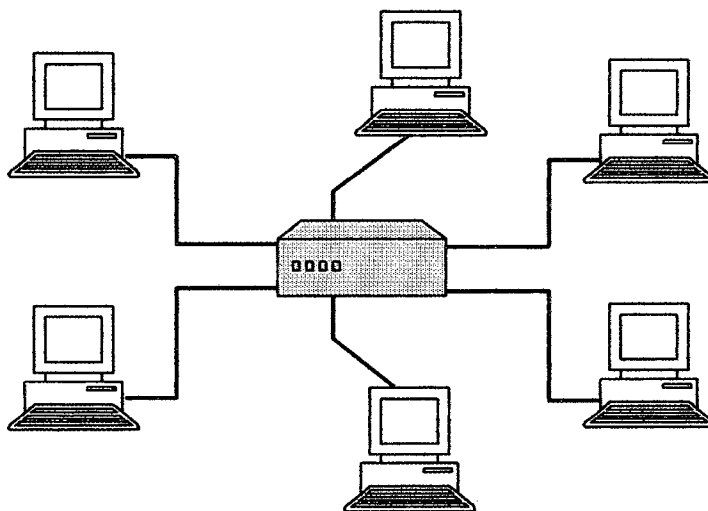


Fig. 3.1 – Topologia em Estrela.

Em algumas redes com topologia em estrela, o nó central tem como única função o gerenciamento das comunicações. Em outras, o nó central tem tanto a função de gerenciamento de comunicações como facilidades de processamento de dados. O nó central é chamado de comutador ou *switch* quando sua função é o chaveamento (ou comutação) entre as estações que desejam se comunicar.

Topologia em Anel

Em uma rede em anel, as estações são conectadas através de um caminho fechado. O anel consiste em uma série de repetidores ligados por um meio físico, e cada estação é ligada à esses repetidores (figura 3.2). As estações, portanto, não são interligadas diretamente ao anel [SOA95].

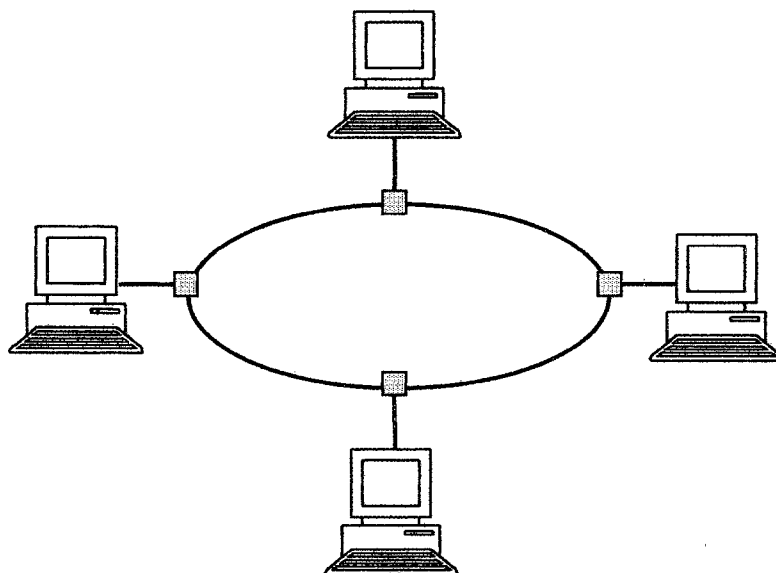


Fig. 3.2 – Topologia em Anel.

A transmissão e recepção de dados em uma rede em anel pode acontecer em ambas as direções. No entanto, as configurações mais usuais são unidirecionais, simplificando o projeto dos repetidores e tornando menos sofisticados os protocolos de comunicação que asseguram a entrega da mensagem ao destino corretamente. Os repetidores são geralmente projetados para transmitir e receber dados simultaneamente, diminuindo o retardo de transmissão.

Quando um nó envia uma mensagem, ela entra no anel e, dependendo do protocolo empregado, circula até que seja retirada pelo nó destino ou até retornar ao nó de origem. No primeiro caso, o repetidor deve usar um retardo suficiente para o recebimento e armazenamento dos bits de endereçamento de destino da mensagem, quando então poderá decidir se esta deve ou não permanecer no anel. No outro caso, a

rede pode atuar com um retardo de um bit por repetidor, porque a medida que os bits de uma mensagem vão chegando eles vão sendo despachados.

Topologia em Barra

Na topologia em barra todas as estações se ligam ao mesmo meio de transmissão (figura 3.3). Nas redes em barra cada nó conectado à barra pode ouvir todas as informações transmitidas.

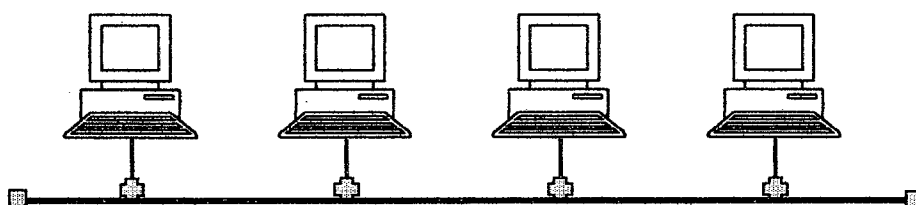


Fig. 3.3 – Topologia em Barra.

Os mecanismos de controle de acesso à barra podem ser do tipo centralizado ou descentralizado. No controle centralizado, o direito de acesso é determinado por uma estação específica da rede. No controle descentralizado, a responsabilidade de acesso é distribuída entre todos os nós.

O meio de transmissão, número de nós conectados, controle de acesso e tipo de tráfego são fatores que determinam o desempenho de um sistema em barra. A interconexão de *hubs* pode ser uma forma de expansão da rede .

3.1.2 Redes Metropolitanas

Uma rede metropolitana (MAN – *Metropolitan Area Network*) é uma versão ampliada de uma LAN porque ambas utilizam tecnologias semelhantes. Uma MAN pode ser privada ou pública e pode abranger um grupo de escritórios vizinhos ou uma cidade inteira [TAN97].

As redes metropolitanas são tratadas como uma categoria especial porque elas tem um padrão especial, o IEEE 802.6, que define dois barramentos aos quais todos os computadores são conectados (figura 3.4).

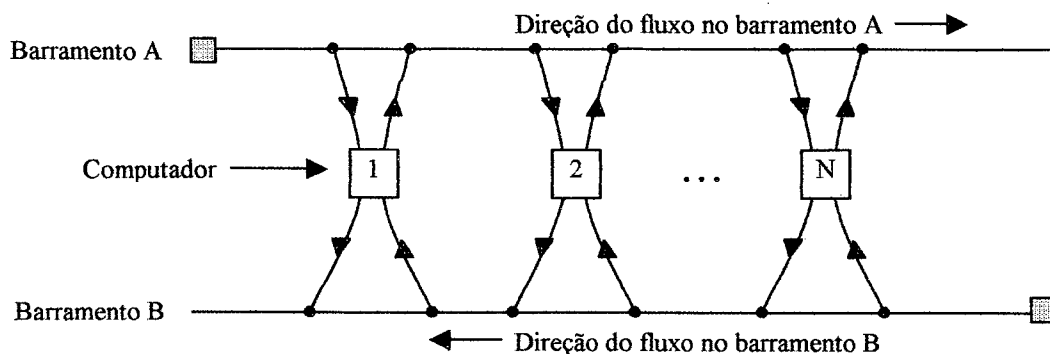


Fig. 3.4 – Arquitetura da rede metropolitana.

Cada barramento tem um dispositivo que inicia a atividade de transmissão. O tráfego que se destina a um computador localizado à direita do emissor utiliza o barramento superior, e o tráfego à esquerda do emissor utiliza o barramento inferior [TAN97].

3.1.3 Redes Geograficamente Distribuídas

Redes Geograficamente Distribuídas (*Wide Area Networks - WANs*) abrangem uma ampla área geográfica, com frequência um país ou continente. Como o custo de comunicação é bastante elevado (circuitos para satélites e enlaces de microondas), tais redes são normalmente gerenciadas por grandes operadoras (públicas ou privadas).

Tendo em vista as considerações em relação ao custo, neste tipo de rede são usados arranjos topológicos específicos e diferentes daqueles utilizados em redes locais para a interligação dos diversos módulos processadores. Caminhos alternativos para a interligação dos diversos módulos processadores devem ser fornecidos, por questões de confiabilidade.

3.2 Arquitetura TCP/IP

O termo arquitetura TCP/IP é utilizado como designação comum para uma família de protocolos de comunicação de dados, sendo que o *Transmission Control Protocol* (TCP) e o *Internet Protocol* (IP) são apenas dois deles. O IP é responsável pelo encaminhamento de pacotes de dados pelas diversas sub-redes desde a origem até seu destino. O TCP tem por função o transporte ponto-a-ponto confiável de mensagens de dados entre dois sistemas. O IP é um protocolo do tipo datagrama, operando, portanto, no modo não orientado à conexão, enquanto o TCP é um protocolo de transporte orientado à conexão. Os protocolos TCP/IP podem, em conjunto, oferecer um serviço confiável. Para uso em redes de alta qualidade, onde a confiabilidade não assume grande importância, foi definido o protocolo UDP (*User Datagram Protocol*) que opera no modo não orientado à conexão e possui funcionalidades bem mais simples que o TCP.

A arquitetura TCP/IP é organizada em camadas como mostra a figura 3.5.

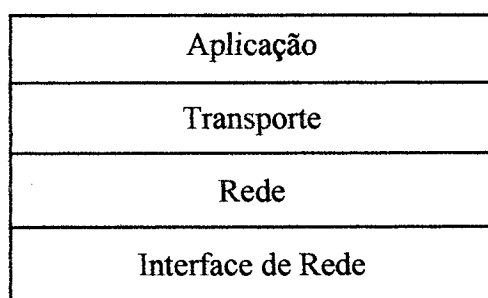


Fig. 3.5 – Organização da arquitetura TCP/IP.

Os protocolos na camada de interface de rede provêm meios para que os dados sejam transmitidos a outros computadores na mesma rede física. As principais funções desta camada são: encapsulamento de datagramas IP em *frames* para transmissão e a tradução de endereços IP em endereços físicos de rede.

A camada de rede é responsável pela transferência de dados através da rede, desde a máquina de origem até a máquina de destino. Esta camada define o protocolo IP, o qual é não orientado à conexão e não é confiável.

A função básica da camada de transporte é permitir a comunicação ponto-a-ponto entre aplicações [SOA95]. Dois protocolos são definidos nesta camada, o TCP e o UDP, que são detalhados nos próximos tópicos.

A camada de aplicação descreve as tecnologias usadas para fornecer serviços especializados para os usuários finais e administra os detalhes de uma aplicação em particular [TEI96]. As aplicações interagem com o nível de transporte para enviar e receber dados.

3.3 TCP (*Transmission Control Protocol*)

O TCP fornece um serviço confiável de transferência de dados e opera no modo orientado à conexão. Ele foi projetado para funcionar com base em um serviço de rede sem conexão e sem confirmação. Desta maneira, ele se responsabiliza pela recuperação de dados corrompidos, perdidos, duplicados ou entregues fora de ordem.

O serviço TCP é obtido quando tanto o transmissor quanto o receptor criam pontos terminais, denominados *sockets*. Um *socket* local pode participar de várias conexões diferentes com *sockets* remotos e para isso utiliza o conceito de porta, onde cada processo que está sendo atendido pelo TCP em um dado momento é identificado por uma porta diferente. Processos servidores que são muito usados como, por exemplo, FTP e Telnet, são associados a portas fixas, as quais são chamadas de portas conhecidas (*well-known ports*).

As conexões no TCP são ponto-a-ponto e transportam fluxos de dados em ambas as direções caracterizando a transmissão full-duplex. O TCP é compatível com os processos de *multicast* e difusão.

Os dados trocados pelas entidades TCP transmissoras e receptoras estão na forma de segmentos. Um segmento consiste em um cabeçalho fixo de 20 bytes (mais uma parte opcional), seguido de zero ou mais bytes de dados [TAN95]. O software TCP define o tamanho destes segmentos e ainda se um segmento vai acumular dados de várias escritas ou dividir os dados de uma única escrita em vários segmentos. Quando um segmento é transmitido, a entidade TCP coloca uma cópia deste segmento em uma fila de retransmissão e aciona um temporizador. Se a entidade transmissora TCP recebe

a confirmação da chegada dos dados, o segmento é retirado da fila. Se o temporizador expirar antes da chegada da confirmação de recebimento dos dados, o segmento é retransmitido. A figura 3.6 mostra o formato do segmento TCP.

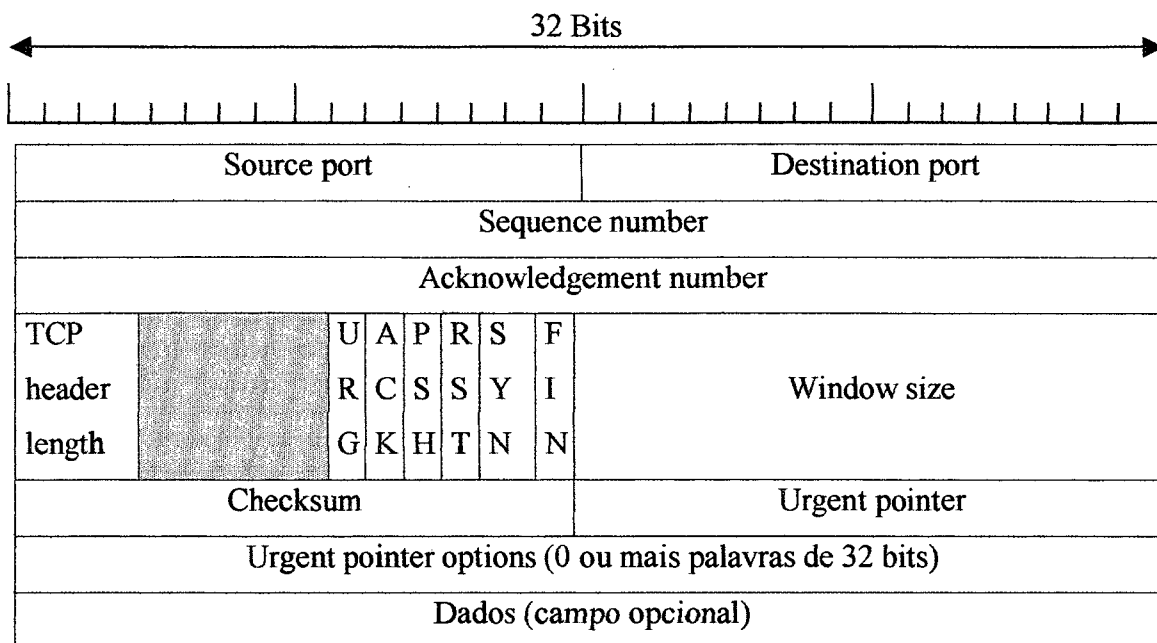


Fig. 3.6 – O segmento TCP.

No TCP receptor, os números de seqüência são usados para ordenar os segmentos que porventura tenham sido recebidos fora de ordem e para eliminar segmentos duplicados [SOA95]. Os segmentos corrompidos são tratados adicionando-se um *checksum* a cada segmento transmitido e no receptor é feita uma verificação sendo que os segmentos danificados são descartados.

Para controlar o fluxo de dados que o transmissor pode enviar ao receptor o TCP usa um mecanismo no qual o receptor informa, juntamente com a confirmação de recebimento, quantos bytes podem ser enviados, contados a partir do último byte confirmado. Com esta informação o transmissor calcula o número de bytes que pode enviar ao receptor.

3.3.2 Estabelecendo uma conexão

O protocolo de estabelecimento de conexão usado no TCP é chamado de *handshake* de três vias (*three-way handshake*). Quando um processo TCP deseja comunicar-se com outro processo TCP, uma seqüência de mensagens é trocada entre os processos. As mensagens trocadas identificam a conexão com números de seqüência baseados em relógios, que são utilizados para evitar que o estabelecimento de conexões inválidas seja provocado pela duplicação de mensagens com pedido de abertura de conexão [SOA95].

Conforme ilustra a figura 3.7, o primeiro segmento de um *handshake* pode ser identificado porque ele possui um conjunto de bits SYN (sincronização) que informam um número de seqüência inicial. A segunda mensagem possui ambos os conjuntos de bits SYN, do processo que iniciou o estabelecimento da conexão, e ACK (confirmação) que contém um número de seqüência inicial do processo receptor, indicando a confirmação de recebimento do primeiro segmento SYN. A mensagem final do *handshake* é uma confirmação utilizada para informar ao processo receptor que ambos os lados concordam em que uma conexão foi estabelecida.

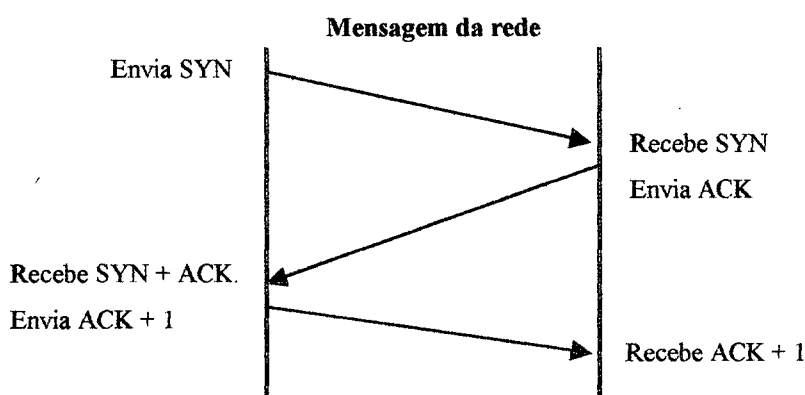


Fig. 3.7 – Handshake de Três Vias.

3.3.3 Encerrando uma conexão

A operação *close* pode ser usada para encerrar uma comunicação entre dois processos que utilizam o TCP. O TCP usa, internamente, um handshake de três vias modificado para encerrar as conexões [COM97]. Para isso o processo TCP transmissor completa a transmissão dos dados restantes, espera que o processo receptor os confirme e, em seguida envia um segmento com o conjunto de bits FIN, que indicam o encerramento da conexão, conforme mostra a figura 3.8. O processo receptor confirma o recebimento do segmento FIN enviando ao processo transmissor um segmento ACK, e em seguida envia o seu segmento FIN. Então, o processo transmissor envia um segmento ACK confirmando que recebeu o segmento FIN do processo receptor.

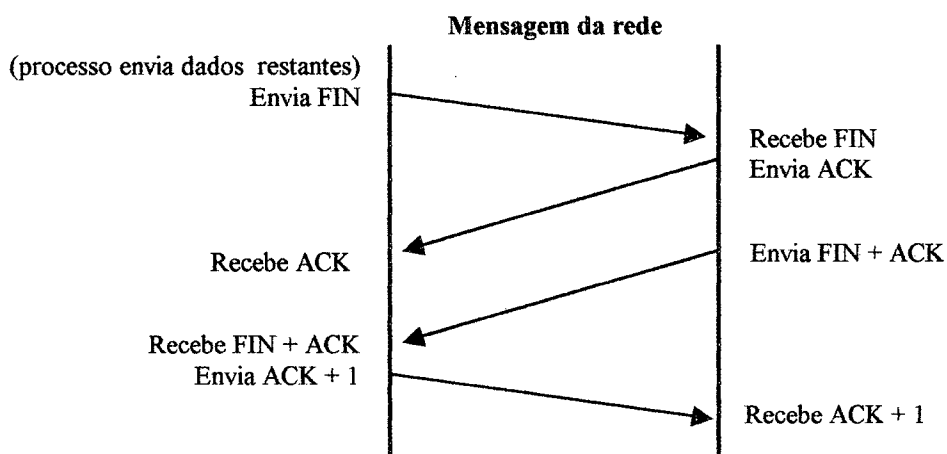


Fig. 3.8 – Handshake de Três Vias utilizado para encerrar conexões.

3.4 UDP (User Datagram Protocol)

O UDP fornece um serviço mínimo de transporte em redes que usam o protocolo IP, permitindo que as aplicações tenham acesso direto aos serviços da camada de rede. O UDP opera no modo sem conexão e fornece um serviço de transporte de dados não-confiável. O fato de ser não orientado à conexão significa que uma mensagem pode ser enviada a qualquer momento, sem qualquer tipo de aviso, negociação ou preparação com a máquina destino. É esperado que a máquina destino

esteja preparada para receber e processar as mensagens, as quais são também chamadas de datagramas. O fato do protocolo não ser confiável significa que :

- Não existe nenhuma garantia de que os datagramas sejam entregues no destino.
- Não existe registro dos datagramas enviados.
- Os datagramas podem chegar fora de ordem.
- Os datagramas podem chegar duplicados.
- Não existe controle de fluxo.
- Não existe controle de congestionamento na rede.

Esta falta de confiabilidade torna necessária a programação, dentro do programa de aplicação, de confirmações de recebimento dos dados por parte do processo receptor, *timeouts* e retransmissões.

O UDP se torna ideal quando não é necessário controle de fluxo nem manutenção das seqüências de mensagens enviadas. Muitas aplicações cliente-servidor que tem uma solicitação e uma resposta utilizam o UDP, como por exemplo: Network File System (NFS), Simple Network Management Protocol (SNMP) e Domain Name System (DNS).

O serviço prestado pelo UDP acrescenta dois serviços que a camada IP não disponibiliza :

- A capacidade de distinguir um entre os vários processos que estejam usando os serviços da camada de rede IP, numa mesma máquina.
- A capacidade de verificação da exatidão dos dados recebidos.

Assim como no TCP, o mecanismo que permite distinguir um entre múltiplos destinos independentes dentro de uma mesma máquina é implementado através do conceito de portas.

O mecanismo que permite verificar se os dados chegaram ao destino intactos é implementado através do campo *checksum* no cabeçalho da mensagem UDP.

Um segmento UDP consiste em um cabeçalho de 8 bytes seguido dos dados, como mostra a figura 3.9.

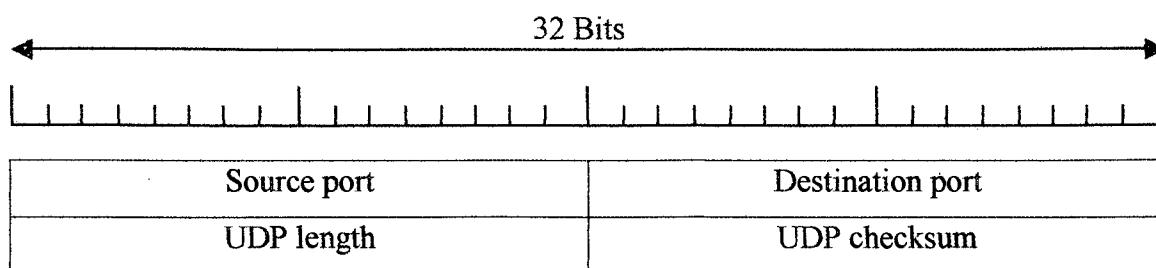


Fig. 3.9 – O segmento UDP.

3.5 Sockets

Sockets constituem uma interface de programação de aplicações (API), e no Unix foram implementados como um conjunto de chamadas de sistema para comunicação entre computadores na arquitetura TCP/IP. A seguir são apresentadas as principais chamadas de sistema relacionadas à criação e manipulação de *sockets* no sistema operacional Unix.

Criação de um socket

A chamada de sistema *socket* cria os *sockets* solicitados. São necessários três argumentos inteiros e é retornado um resultado inteiro. Sua forma é a seguinte:

resultado = **socket** (*pf*, *tipo*, *protocolo*);

O argumento *pf* especifica a família de protocolos a ser usada com o *socket*. Os principais são AF_INET (para protocolo IPv4), AF_INET6 (para protocolo IPv6) e AF_LOCAL (para protocolos *Unix Domain*). O argumento *tipo* especifica o tipo de comunicação a ser usada. Entre os tipos possíveis estão o serviço de transmissão confiável (SOCK_STREAM) e o serviço de transmissão de datagrama sem conexão (SOCK_DGRAM), assim como um tipo bruto (SOCK_RAW) que permite que programas privilegiados acessem protocolos de nível inferior ou interfaces de redes. O argumento *protocolo* geralmente é setado para zero, exceto para *sockets raw*. Quando executada com sucesso, a chamada de sistema *socket* retorna um valor inteiro, não-negativo, chamado descritor do *socket* [COM98]. Senão, o valor de retorno é -1.

Fechamento de um socket

Quando um processo acaba de utilizar um *socket*, ele invoca a chamada de sistema *close* que tem a seguinte forma:

close (sockfd);

O argumento *sockfd* especifica o descritor de *socket* a ser fechado. Internamente, uma chamada *close* decrementa a contagem de um *socket* e o destrói se a contagem chegar a zero [COM98]. Quando executada com sucesso, a chamada de sistema *close* retorna zero, senão retorna -1 [STE97].

Especificação de um endereço local

Uma vez criado um *socket*, é possível associar um endereço (TCP, UDP ou IP) a ele através da chamada de sistema *bind* que tem o seguinte formato:

bind (socket, localaddr, addrlen);

O argumento *socket* é o descritor do *socket* a ser vinculado. O argumento *localaddr* é um endereço de memória para a estrutura que especifica o endereço local ao qual o *socket* deve ser vinculado e o argumento *addrlen* é um inteiro que especifica o comprimento do endereço medido em bytes. Quando executada com sucesso, a chamada de sistema *bind* retorna zero, senão retorna -1 [STE97].

Vinculação de sockets a endereços destinos

Um *socket* é criado no estado desconectado, isto é, ele não está associado a nenhum endereço destino. A chamada de sistema *connect* vincula um destino permanente a um *socket*, colocando-o para o estado conectado. O formato desta chamada de sistema é o seguinte:

connect (socket, destaddr, addrlen);

O argumento *socket* é o descritor do *socket* a ser conectado. O argumento *destaddr* é um endereço de memória para uma estrutura de *socket* que indica o endereço de destino ao qual o *socket* deve ser vinculado. O argumento *addrlen* indica o tamanho, em bytes, da estrutura de endereço de *socket*. Quando executada com sucesso,

a chamada de sistema *connect* retorna zero, senão retorna -1 [STE97].

Envio de dados através de um socket

Existem cinco chamadas de sistema possíveis de serem usadas para transmitir dados: *send*, *sendto*, *sendmsg*, *write* e *writetv*. *Send*, *write* e *writetv* funcionam somente com *sockets* conectados porque não permitem que o processo solicitante especifique um endereço de destino. A chamada de sistema *send* tem a seguinte forma:

send (*socket*, *message*, *length*, *flags*);

O argumento *socket* especifica o *socket* a ser usado, o argumento *message* fornece o endereço dos dados a serem enviados e o argumento *length* especifica o número de bytes a serem enviados. O argumento *flags* controla a transmissão, permitindo, por exemplo, que o solicitante determine que a mensagem seja enviada sem usar as tabelas de roteamento locais. Quando executada com sucesso, a chamada de sistema *send* retorna o número de bytes transmitidos, senão retorna -1.

Recebimento de dados através de um socket

Assim como para o envio, existem cinco chamadas de sistemas que podem ser utilizadas para recebimento de dados: *read*, *readv*, *recv*, *recvfrom*, e *recvmsg*.

Os processos usam a chamada de sistema *recv* para receber dados de um *socket* conectado. Sua forma é a seguinte:

recv (*socket*, *buffer*, *length*, *flags*);

O argumento *socket* especifica um descritor de *socket* do qual os dados devem ser recebidos. O argumento *buffer* especifica o endereço da memória na qual a mensagem deve ser colocada e o argumento *length* especifica o comprimento da área de buffer. O argumento *flags* permite que o processo solicitante controle a recepção dos dados [COM98]. Quando executada com sucesso, a chamada de sistema *recv* retorna o número de bytes recebidos, senão retorna -1.

Especificação de um comprimento de fila para um servidor

A chamada de sistema *listen* é usada por processos servidores para preparar um *socket* para conexões de entrada. Um processo servidor chama *listen* para informar ao sistema operacional que o software do protocolo deve enfileirar as várias solicitações simultâneas que chegam ao *socket*. O formato desta chamada de sistema é o seguinte:

listen (*socket*, *qlength*);

O argumento *socket* especifica o descritor de *socket* a ser usado por um servidor. O argumento *qlength* especifica o comprimento da fila de solicitação para aquele *socket*. Se a fila estiver cheia quando uma solicitação chegar, o sistema operacional recusa a conexão e descarta a solicitação. Esta chamada de sistema é usada apenas por *sockets* que selecionaram um serviço de entrega confiável. Quando executada com sucesso, *listen* retorna zero; senão retorna -1 [STE97].

Aceitando conexões

A chamada de sistema *accept* é utilizada por processos servidores que desejam aguardar uma conexão. Uma chamada à *accept* permanece bloqueada até que uma solicitação de conexão chegue. Seu formato é o seguinte:

newsock = ***accept*** (*socket*, *addr*, *addrlen*);

O argumento *socket* especifica o descritor de *socket* a ser usado. O argumento *addr* é um ponteiro para um endereço de memória para uma estrutura do tipo *sockaddr* e o argumento *addrlen* é um ponteiro para um inteiro. Quando uma solicitação chega, o sistema preenche o argumento *addr* com o endereço do cliente que fez a solicitação e retorna um novo descritor de *socket* ao solicitante. O *socket* original permanece aberto e o processo servidor pode continuar recebendo solicitações. Quando executada com sucesso, a chamada de sistema *accept* retorna um novo descritor de *socket*; senão retorna -1 .

3.6 Servidores Concorrentes

Servidores concorrentes com *sockets* TCP/IP [STE98] constituem uma técnica muito utilizada na criação de processos servidores. Nesta técnica, o servidor fica a espera de novas conexões através do *socket listenfd*, como mostra a figura 3.10 (a). Quando uma requisição de conexão chega ao servidor, ela é imediatamente aceita pelo núcleo do sistema operacional e um novo *socket, connfd*, é criado, como mostra a figura 3.10 (b). Em seguida, o processo servidor executa um *fork* criando um processo servidor filho para atender esta requisição, conforme ilustra a figura 3.10 (c). Ambos os *sockets, listenfd* e *connfd*, são compartilhados entre os processos pai e filho. A figura 3.10 (d) apresenta o estado final dos *sockets*, onde o processo servidor pai fecha o *socket connfd* e continua esperando requisições de conexões no *socket listenfd* e o processo servidor filho fecha o *socket listenfd* e atende a requisição pelo *socket connfd*.

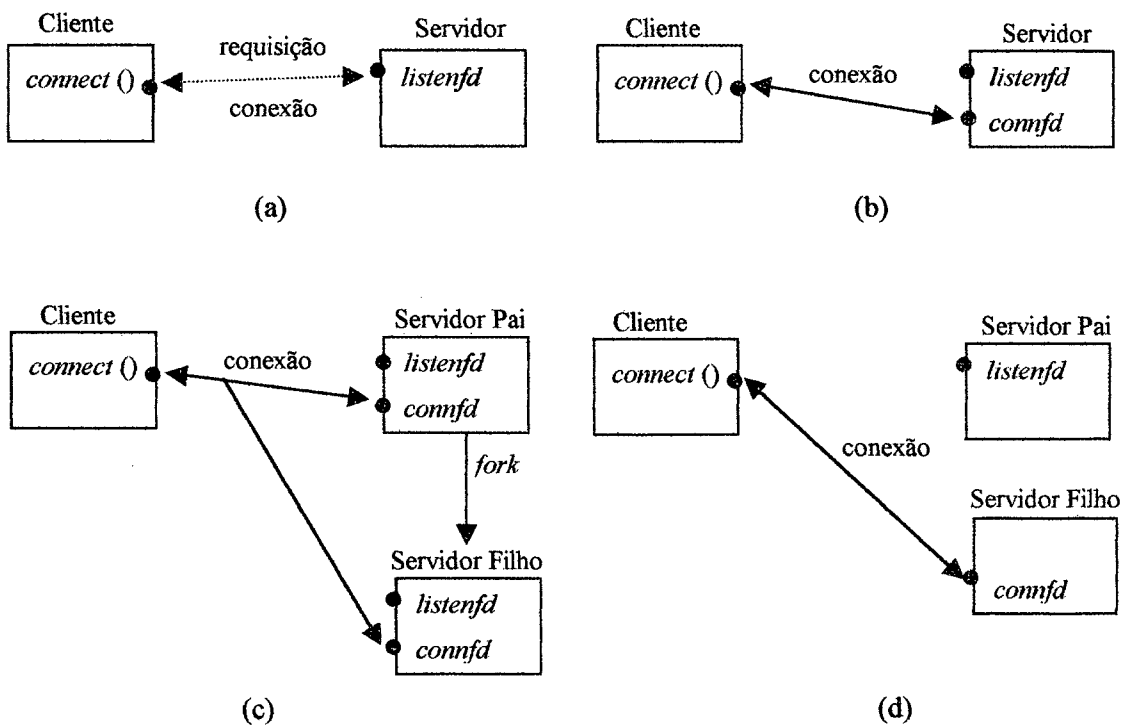


Fig. 3.10 – Servidor concorrente com *sockets* TCP/IP.

4. Sistemas Computacionais com Memória Distribuída

Neste capítulo, são apresentados os multicomputadores e os clusters de computadores. No tópico 4.1 são apresentados os principais conceitos relacionados aos multicomputadores e no tópico 4.2 são introduzidos os clusters de computadores.

4.1 Multicomputadores

No subtópico 4.1.1 é exposta uma classificação de arquiteturas de computadores para posicionar os multicomputadores nesse contexto, no subtópico 4.1.2 são descritas as redes de interconexão para multicomputadores e no subtópico 4.1.3 é apresentado o ambiente multicomputador Crux.

4.1.1 Classificação

Do ponto de vista da arquitetura de computadores, é possível situar os multicomputadores através da classificação de Flynn (1972), a qual se baseia sobre duas características: o número de fluxos de instruções e o número de fluxos de dados. A classificação está dividida em quatro classes, a saber:

- SISD (*Single Instruction Stream - Single Data Stream*): existe um único fluxo de instruções que opera sobre um único fluxo de dados. Nesta classe estão incluídas as máquinas seqüenciais convencionais, onde existe somente uma unidade de controle que decodifica seqüencialmente as instruções, que por sua vez manipulam um conjunto de dados.
- SIMD (*Single Instruction Stream - Multiple Data Stream*): existe um único fluxo de instruções que opera sobre múltiplos fluxos de dados. Este modelo é adequado para aplicações envolvendo cálculos matriciais intensivos.

- MISD (*Multiple Instruction Stream - Single Data Stream*): existem múltiplos fluxos de dados autônomos de instruções que operam sobre um fluxo de dados único. Nenhum computador conhecido segue esse esquema, sendo portanto, um esquema teórico.
- MIMD (*Multiple Instruction Stream - Multiple Data Stream*): existem múltiplos fluxos autônomos de instruções que operam sobre múltiplos fluxos de dados. Os computadores paralelos de uso geral se enquadram nesta classe.

De forma complementar, um segundo nível de classificação pode ainda ser feito, apresentando os computadores da arquitetura MIMD divididos em dois grupos: aqueles que possuem memória compartilhada, chamados de multiprocessadores, e aqueles que não possuem memória compartilhada, chamados de multicomputadores.

- Multiprocessadores: existe um único espaço de endereçamento virtual que é compartilhado por todos os processadores. Todas as máquinas compartilham a mesma memória.
- Multicomputadores: cada máquina tem sua memória privativa. A interação se dá através de troca de mensagens pela rede de interconexão.

4.1.2 Redes de Interconexão

Redes de interconexão são constituídas de entidades de hardware (canais de comunicação) e software (controle do estabelecimento dos canais) que são projetadas para facilitar a comunicação entre processos e processadores [MER96].

A topologia de uma rede de interconexão pode ser estática ou dinâmica. Em uma rede estática os elementos são conectados por meio de ligações ponto-a-ponto fixas, que não mudam durante a execução do programa. Uma rede dinâmica é formada por canais chaveados que são configurados dinamicamente, conforme a demanda do programa em execução.

Redes Estáticas

Multicomputadores com redes de interconexão estáticas apresentam canais ligando direta e estaticamente os nós sem a possibilidade de reconfiguração [MER96]. A comunicação entre dois nós que não estejam ligados diretamente se dá por intermédio de outros nós.

Várias estruturas de redes estáticas são utilizadas em multicomputadores e a seguir são mostradas algumas delas.

A *grelha* quadrada consiste de uma coleção de nós conectados como mostra a figura 4.1. É caracterizada por nós interiores com quatro vizinhos. Diversos processadores matriciais, como o Illiac IV [COR99], utilizam esse tipo de rede.

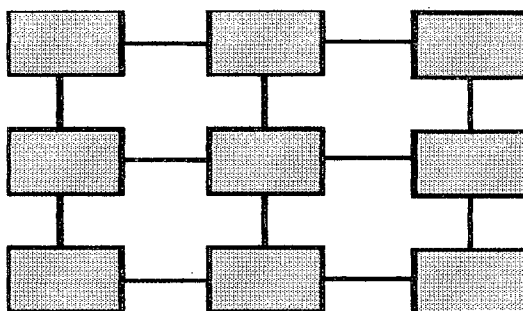


Fig. 4.1 – Grelha.

O *hipercubo* constitui uma rede de interconexão estática de dimensão k , tendo 2^k nós interconectados de forma que cada nó é ligado a k nós vizinhos. A figura 4.2 mostra o formato de uma rede hipercúbica de dimensão três.

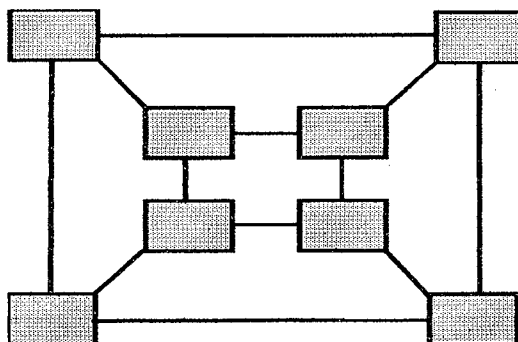


Fig. 4.2 – Hipercubo.

O *torus* é uma variante da grelha, obtida pela interligação dos nós externos das mesmas linhas e colunas, conforme mostra a figura 4.3. O grau dos nós do torus é constante.

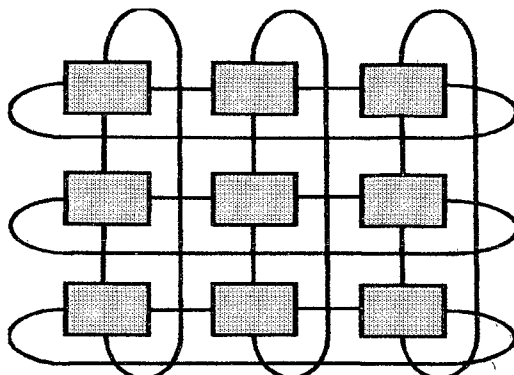


Fig. 4.3 – Torus.

Redes Dinâmicas

As redes dinâmicas apresentam estrutura reconfigurável conforme a demanda do programa em execução e são mais utilizadas em aplicações de propósito geral onde os padrões de comunicação são imprevisíveis. Topologias desse tipo de rede são classificadas em barramentos, redes multiestágio e *crossbar*.

O *barramento* é formado por um conjunto de fios e conectores que estabelecem um meio de comunicação comum a todos os elementos por ele conectados, conforme a figura 4.4. Possui baixo custo e limitação na sua capacidade de transferência, acarretando a degradação de desempenho do sistema quando sobrecarregado.

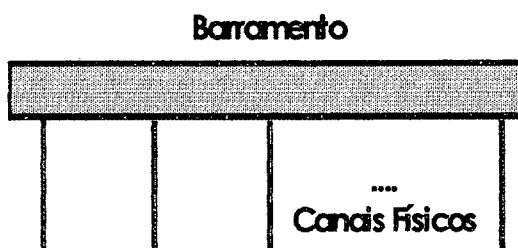


Fig. 4.4 – Barramento.

Redes multiestágio 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 entrada-saída. Vários estágios aumentam o tempo para conectar dois nós, o que se torna significativo em redes de muitos estágios. Um exemplo de rede multiestágio é a rede ômega como mostra a figura 4.5.

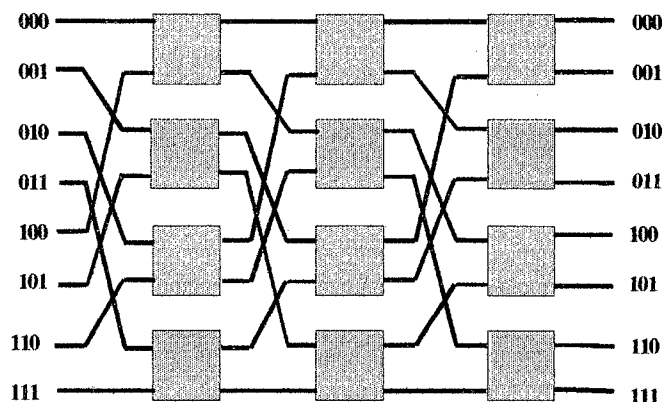


Fig. 4.5 – Rede Ômega 8 x 8.

Um *crossbar* é uma rede de interconexão que possui um ponto de cruzamento para uma entrada e uma saída como mostra a figura 4.6. Apenas um ponto de contato precisa ser fechado para estabelecer a conexão de qualquer par de nós.

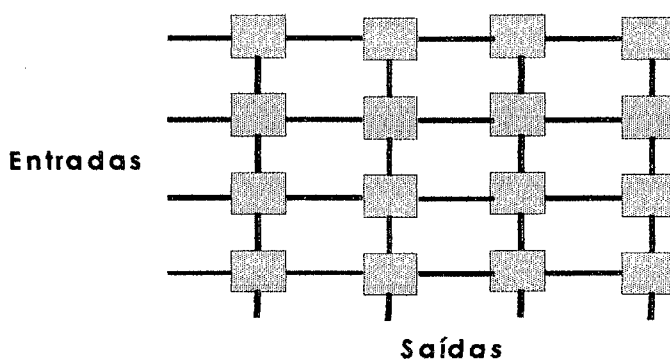


Fig. 4.6 – Crossbar 4 x 4.

4.1.3 Ambiente Multicomputador Crux

O multicomputador Crux visa disponibilizar um ambiente para execução de programas paralelos expressos como redes de processos comunicantes. A seguir é apresentada a arquitetura desse ambiente multicomputador.

Arquitetura

O multicomputador constitui-se por um conjunto de *nós de trabalho*, os quais possuem processadores com memória privativa e vários canais físicos. Esses nós executam os processos das redes de processos comunicantes. Existe também uma *rede de trabalho*, uma *rede de controle* e um *nó de controle* que são explicados a seguir.

A rede de trabalho (um crossbar) faz o transporte de mensagens entre dois pares de nós de trabalho quaisquer. Esse transporte é efetuado através de canais físicos diretos.

O nó de controle, através de um processo específico, configura a rede de trabalho conforme a demanda do programa paralelo em execução nos demais nós e é responsável também pela atribuição/liberação de nós de trabalho.

A rede de controle, como o próprio nome indica, transporta mensagens de controle entre os nós de trabalho e o nó de controle. A figura 4.7 apresenta a arquitetura do multicomputador Crux.

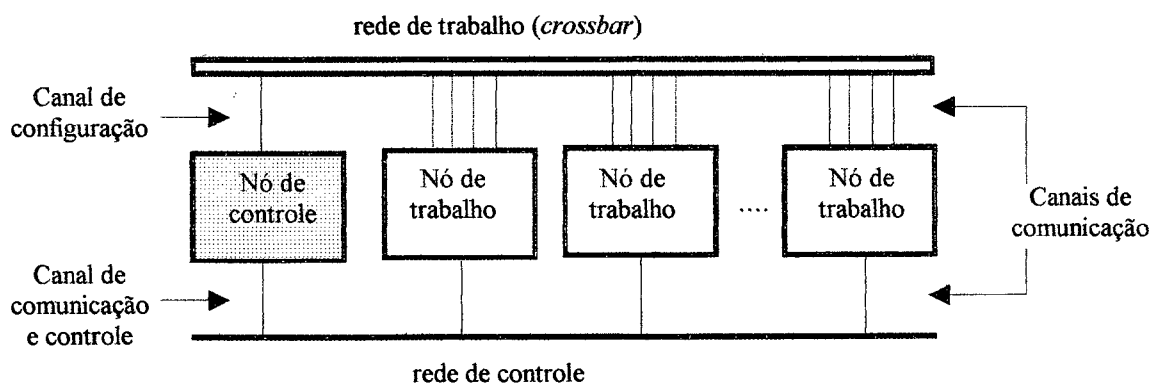


Fig. 4.7 – Arquitetura do multicomputador Crux.

4.2 Cluster de Computadores

No subtópico 4.2.1 é apresentada uma visão geral sobre o assunto, no subtópico 4.2.2 são mostrados os mecanismos de comunicação em clusters e no subtópico 4.2.3 são descritos projetos que envolvem a montagem e utilização de clusters.

4.2.1 Visão Geral

Clusters de computadores vem ganhando crescente destaque como ambiente para processamento paralelo. Basicamente, um *cluster* é uma coleção de computadores pessoais ou estações de trabalho conectados por uma rede, montados com componentes comerciais [RIB01].

As principais alternativas para processamento paralelo de alto desempenho oferecida até recentemente eram os multiprocessadores e os multicomputadores. Essas máquinas apresentam preços muito elevados e altas taxas de obsolescência, agravadas pelo rápido desaparecimento do mercado de modelos e fabricantes. O expressivo aumento da confiabilidade e do desempenho de computadores pessoais, estações de trabalho e componentes de redes de computadores viabilizou o uso de clusters como uma alternativa de baixo custo para processamento paralelo em aplicações que exigem muitos recursos computacionais.

A programação paralela em um *cluster* pode se apoiar em sistemas operacionais como o Linux e pacotes de programação paralela como PVM (*Parallel Virtual Machine*) [PVM01] ou MPI (*Message Passing Interface*) [MPI01].

A organização física de um cluster normalmente é composta pelos computadores interligados por *hubs* ou *switches*, como mostra a figura 4.8.

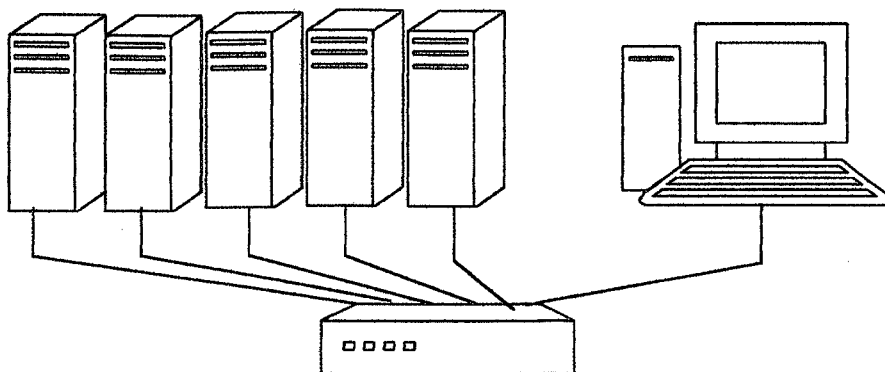


Fig. 4.8 – Estrutura genérica de um cluster.

4.2.2 Sistema Operacional em Clusters

De modo geral, o sistema operacional de um cluster possui objetivos similares aos de uma estação de trabalho [FER01]. Estes objetivos são os de escalonar os múltiplos processos dos usuários num conjunto único de componentes de hardware (gerenciamento de recursos, tendo como ênfase os múltiplos processadores e memórias), e os de prover abstrações para o software de alto nível. As abstrações incluem a proteção de acesso do usuário a componentes críticos, processos e mecanismos de comunicação.

Existem diversas características desejáveis em um sistema operacional de clusters, tais como:

- **Transparência:** O usuário deve ter a visão de uma única máquina, ao invés de diversas máquinas interconectadas;
- **Estabilidade:** O ambiente de execução deve ser robusto porque as aplicações executadas nos clusters, pela sua complexidade, costumam ser de longa duração.
- **Desempenho:** O desempenho é muitas vezes um critério essencial para um sistema operacional de cluster, pois os clusters são construídos para a obtenção de alto desempenho.

- Escalabilidade: Um cluster deve ter a capacidade de ser aumentado pela inclusão de novos nós para aumentar o desempenho global.

A forma de se obter as características descritas anteriormente é o aproveitamento de um sistema operacional convencional estendido por uma camada de software. Essa camada deve oferecer serviços capazes de satisfazer as principais necessidades da programação paralela.

4.2.3 Mecanismos de Comunicação em Clusters

A comunicação é um fator fundamental no desempenho dos clusters. A combinação de diferentes tecnologias de hardware e software nessa área determinam o desempenho global do cluster. A seguir, é feita uma abordagem das principais tecnologias de hardware e de software para comunicação de clusters.

Tecnologias de Hardware

Como tecnologias de hardware utilizada em clusters, pode-se citar:

HiPPI

O HiPPI (*High Performance Parallel Interface*) [TAN97] fornece grande largura de banda para transferências de dados. Essa tecnologia foi projetada com a intenção de fornecer um meio de comunicação entre máquinas com grandes exigências de E/S, como os supercomputadores. O projeto desta tecnologia inclui a utilização de comutadores *crossbar*, permitindo a construção de redes locais de alta velocidade.

HiPPI seriais vem tornando-se populares com o uso de fibra ótica, porém ainda são caras.

SCI

A tecnologia SCI (*Scalable Coherent Interconnect*) tem por objetivo prover um mecanismo de alto desempenho que suporte com coerência o acesso à memória compartilhada através de um grande número de máquinas.

Em SCI, a transferência de dados dá-se por comunicação implícita através de acessos remotos à memória. Cada nó pode mapear para seu próprio espaço de endereçamento segmentos remotos de memória pertencentes a qualquer outro nó, atuando como se tais segmentos fossem locais. A comunicação real é levada a efeito de forma transparente pelo hardware SCI — por exemplo, placas de rede —, que se responsabiliza pelas leituras e escritas em segmentos remotos [CIA99].

Myrinet

Myrinet é uma tecnologia de rede de alta velocidade que fundamenta-se na comunicação por troca de mensagens. Foi desenvolvida para se tornar uma tecnologia de interconexão de baixo custo baseada em chaveamento e comunicação por pacotes, principalmente para a interconexão de clusters de computadores de diferentes tipos.

As características que distinguem a Myrinet das demais tecnologias são: portas e interfaces *full-duplex* alcançando 1.28 Gbps para cada *link*; controle de fluxo, de erro, e monitoramento contínuo dos *links*; baixa latência, *switches* crossbar com monitoramento para aplicações de alta disponibilidade; suporte a qualquer configuração de topologia; interfaces das estações possuem programa de controle para interagir diretamente com os processos para realizar comunicação com baixa latência [MYR01].

Fast Ethernet

A Ethernet é a tecnologia mais utilizada em redes locais, tendo sido especificada pela norma IEEE 802.3. Uma rede Ethernet permite normalmente velocidades de até 10 Mbps na transmissão dos pacotes. O Fast Ethernet é a migração

desta tecnologia para a velocidade para 100 Mbps. As demais características do padrão Ethernet, tais como o formato do *frame*, a quantidade de dados que um *frame* pode conter e o mecanismo de controle de acesso ao meio são mantidas [COM98].

Técnicas de Software

Em clusters de computadores, que são arquiteturas com memória distribuída, a troca de mensagens é a forma natural de comunicação entre processos. A seguir são mostrados as principais técnicas de software para comunicação em clusters.

Mensagens Ativas

Mensagens ativas [CIA99] constituem uma forma derivada do modelo clássico de envio e recebimento de mensagens. O objetivo desse mecanismo é reduzir o *overhead* da comunicação sobre o desempenho das aplicações.

Cada mensagem que chega no processo destino ativa uma função criada pelo programador. Essa função atua como um fluxo de execução que imediatamente processa a mensagem e informa ao fluxo de execução principal sobre a sua chegada. Isto elimina a necessidade de grande quantidade de armazenamento temporário para mensagens, aumentando consideravelmente a velocidade nas comunicações.

Atualmente muitos sistemas exploram mecanismos de comunicação baseados no modelo de mensagens ativas [CIA99].

Chamada de Procedimento Remoto

Chamadas de procedimentos remotos constituem um padrão bastante difundido de comunicação em sistemas distribuídos cliente-servidor. Essa tecnologia provê uma estrutura para o projeto de aplicações cliente-servidor onde os serviços remotos são visualizados como procedimentos. Tais serviços são requisitados pelos clientes através da chamada de procedimentos do servidor com parâmetros adequados.

O serviço chamado pode também retornar um resultado. Esta semântica é vista pelos programadores como muito similar a programação seqüencial [DIE01].

PVM

PVM (*Parallel Virtual Machine*) fornece uma interface de programação para controle de processos e troca de mensagens. O objetivo do PVM é fornecer um ambiente de programação paralela e distribuída, onde os programas são executados em máquinas heterogêneas interligadas. O PVM foi inicialmente implementado em máquinas Unix usando o protocolo TCP/IP. Atualmente existem versões para diversos outros sistemas, garantindo larga portabilidade.

O PVM utiliza o conceito de máquina paralela virtual. Esta máquina virtual é formada por um conjunto de máquinas físicas interligadas, usadas pelos usuários como uma máquina paralela para execução de programas paralelos. Os processos de um programa paralelo se comunicam por troca de mensagens [PVM01].

MPI

MPI (*Message Passing Interface*) inclui a definição de interfaces de um conjunto de rotinas para comunicação por troca de mensagens. O MPI propõe uma padronização para a interface de troca de mensagens para máquinas paralelas com memória distribuída, a fim de aumentar a portabilidade dos programas entre as diferentes máquinas.

O núcleo do MPI é formado por rotinas de envio e recepção de mensagens entre processos. O MPI define ainda várias outras rotinas, entre as quais se destacam rotinas de comunicação em grupo. Essas rotinas permitem, por exemplo, envio de um mesmo dado de um processo participante do grupo a todos os demais processos do grupo [MPI01].

O paralelismo no MPI é explícito, o programador é responsável em identificar o paralelismo e implementar um algoritmo utilizando construções com o MPI.

4.2.4 Projetos de Clusters

São apresentados a seguir alguns projetos envolvendo o uso de clusters.

Parnass2

Objetivo

O Parnass2 [SCH01] é um cluster de computadores pessoais usado para pesquisas de computação científica no Departamento de Matemática Aplicada da Universidade de *Bonn*. O principal objetivo do Parnass2 é servir como plataforma para atividades de computação que exigem alto desempenho.

Nós

O cluster Parnass2 possui 72 nós. Cada nó consiste de 2 processadores Pentium II de 400 MHz, 256 MBytes de memória principal, 8.4 GBytes de memória em disco e estão conectados pelas redes Myrinet e Fast Ethernet. Além disso, um nó adicional é usado como servidor de login e um outro como servidor NFS

Rede

O Parnass2 emprega seis *switches*, doze adaptadores para redes locais Myrinet e trinta e seis adaptadores SAN (*System Area*) para Myrinet. Estes componentes são conectados em uma topologia de três níveis.

Sistema Operacional e Software de Comunicação

O sistema operacional utilizado no cluster é o Linux com suporte a SMP (*symmetrical multi processor*).

O sistema de troca de mensagens utilizado é o *Score 2.4 / MPICH-PM*. O *Score/PM* é um software que fornece um ambiente de programação paralela para clusters de computadores, desenvolvido pelo Laboratório de Software para Sistemas Distribuídos e Paralelos – Japão. O *MPICH* é uma implementação portátil do MPI que fornece grande desempenho em clusters de computadores e foi desenvolvida pelo Laboratório Nacional de *Argonne* – Universidade de Chicago. Um dos softwares disponíveis no cluster é o Scalapack, uma extensão do Lapack (*Linear Álgebra Package*) para programação paralela.

The Texas Tech Tomado Cluster

Objetivo

O t3c (*Texas Tech Tornado Cluster*) [TEX01] tem por objetivo prover aos alunos da Universidade do Texas uma ferramenta para pesquisa sobre processamento paralelo. Com a intenção de verificar se um cluster montado com estações de trabalho e componentes comerciais proporciona um ambiente paralelo apropriado para o aprendizado de programação e processamento paralelo, vem sendo desenvolvidos estudos com os estudantes dessa Universidade a fim de melhor entender como montar, configurar e gerenciar um cluster.

Nós

O cluster t3c possui 22 nós. Os nós possuem processadores Pentium com velocidades que variam de 40 à 166 MHz e memória principal com capacidade de

armazenamento variando de 8 à 64 Mbytes. Alguns nós que compõem este cluster são estações de trabalho Sun Sparc.

Rede

A interconexão da rede é feita por um *switch* Ethernet que atua como um crossbar e pode ser configurado de várias formas em relação à transferência de pacotes.

Sistema Operacional e Software de Comunicação

O sistema operacional utilizado nos nós do cluster t3c é o Linux. A instalação foi feita como se cada nó do cluster fosse um sistema multi-usuário independente.

As ferramentas de desenvolvimento de software paralelo LAM/MPI também foram instaladas. O LAM (*Local Area Multicomputer*) é um tipo de implementação de MPI.

GAMMA

Objetivo

O GAMMA (*Genoa Active Message Machine*) [CIA99] é um sistema de transferência de mensagens para clusters de computadores que executam o sistema operacional Linux e que usam o *Fast Ethernet* como tecnologia de rede. Todas as funcionalidades multi-usuário e multi-tarefa do núcleo do Linux foram preservadas para aplicações seqüenciais e estendidas para aplicações paralelas.

Nós

O sistema GAMMA está apto a funcionar em clusters formados por computadores com componentes comerciais e placas de rede Ethernet.

Rede

A tecnologia de rede utilizada é Fast Ethernet. O protocolo de comunicação implementado neste sistema é do tipo não-confiável e com isso os erros de comunicação (pacotes perdidos e pacotes corrompidos) são detectados mas não são tratados. Dessa maneira, fica a cargo do usuário a tarefa de usar mecanismos de manipulação de erros.

O *driver* de dispositivo do GAMMA está habilitado a gerenciar comunicações IP padrão e comunicações GAMMA simultaneamente, separando pacotes IP dos pacotes GAMMA. Os pacotes IP são repassados para o código específico correspondente no núcleo do Linux e então processados.

Sistema Operacional e Software de Comunicação

Um dos requerimentos do sistema GAMMA é que o sistema operacional dos nós do cluster seja o Linux. Os pacotes de programação paralela PVM e MPI também podem estar presente.

A inovação deste projeto está na aplicação do paradigma de comunicação de mensagens ativas para clusters de computadores. Os mecanismos básicos de comunicação do sistema GAMMA são implementados como um pequeno conjunto de chamadas de sistemas colocados no núcleo do Linux.

Um programa paralelo GAMMA é um grupo de processos contendo N instâncias de processos executando em paralelo, cada instância executando em um nó distinto. Atualmente, um grupo de processos não inclui mais processos que o número de nós que formam o cluster. Assim, cada instância de processo executa em um nó distinto. Cada grupo de processos é identificado por um número único, chamado de PID

paralelo. Processos que pertencem ao mesmo grupo compartilham o mesmo PID paralelo. Este PID paralelo serve para diferenciar os processos que pertencem a diferentes aplicações paralelas

O GAMMA tem suporte para multiprogramação paralela, onde mais de um grupo de processos pode estar ativo no cluster, ao mesmo tempo. Cada nó compartilha o tempo do processador entre processos pertencentes a diferentes grupos de processos.

O GAMMA usa o conceito de Portas Ativas, onde cada porta pode ser usada para trocar mensagens com outros processos do mesmo grupo de processos, ou de diferentes grupos. Cada porta pode ser usada para entrada, saída ou entrada/saída. As portas de saída são usadas para enviar mensagens. As portas de entrada são usadas para receber mensagens e as portas de entrada/saída são usadas para envio e recebimento de mensagens. Antes de usar uma porta, deve ser definido seu comportamento para entrada, saída ou entrada/saída. Isto é feito através de uma função fornecida pela biblioteca GAMMA.

4.2.5 Tabela Comparativa

A tabela 4.1 resume as principais características dos projetos citados acima.

Tabela 4.1 - Principais características dos projetos.

Projeto	Tecnologia de Rede		Sistema Operacional	Softwares Disponíveis		
	<i>Myrinet</i>	<i>Fast Ethernet</i>	<i>Linux</i>	<i>PVM</i>	<i>MPI</i>	<i>Outros</i>
Parnass2	X	X	X			Scalapack
Tornado		X	X			LAM/MPI
GAMMA		X	X	X	X	

5. Servidor de Arquivos

Este capítulo tem por objetivo principal apresentar o servidor de arquivos desenvolvido para um cluster derivado do multicomputador Crux. Inicialmente, no tópico 5.1 é descrito o cluster alvo. No tópico 5.2 é apresentado o modelo operacional deste servidor de arquivos e no tópico 5.3 são mostrados os detalhes de sua implementação. No tópico 5.4 é apresentado, através de um exemplo, as etapas de interação entre um processo cliente e o servidor de arquivos.

5.1 Cluster Alvo

O cluster alvo é derivado do multicomputador Crux apresentado em 4.1.3. Na arquitetura genérica do cluster alvo mostrada na figura 5.1, um computador representa o nó de controle e os demais computadores representam os nós de trabalho. O *hub* representa a rede de controle e os *switchs* representam a rede de trabalho. O sistema operacional utilizado é o Linux e o protocolo para comunicação entre esses computadores é o TCP/IP.

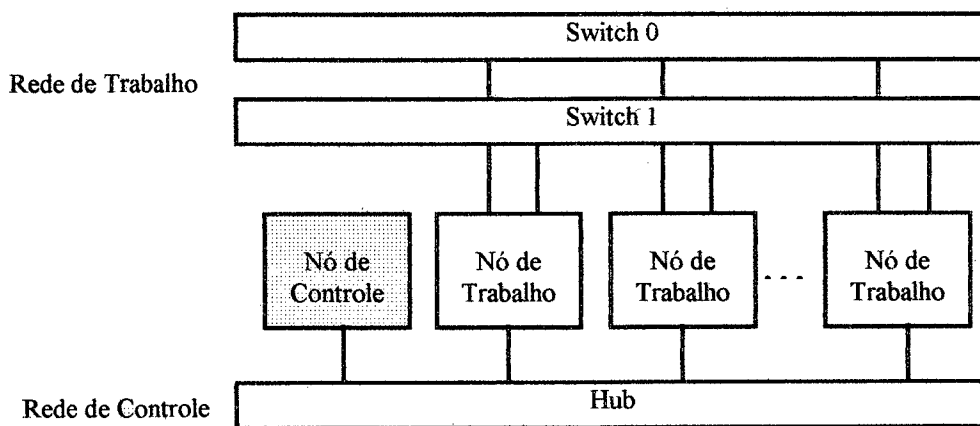


Fig. 5.1 – Arquitetura genérica do cluster alvo.

5.2 Modelo Operacional

O objetivo deste trabalho consiste no projeto e implementação de um servidor de arquivos para o cluster alvo. A função desse servidor é oferecer aos processos clientes os serviços de acesso a informações armazenadas em discos magnéticos. O servidor de arquivos para o cluster alvo deve atender requisições de processos clientes independentes usando o mecanismo de troca de mensagens específico desse cluster.

Para permitir o desenvolvimento concomitante deste trabalho com o das interfaces da rede de controle e da rede de trabalho, que são objeto de outras dissertações de mestrado [REC02, BOG02], decidiu-se trabalhar sobre uma rede conforme a figura 5.2 e implementar provisoriamente a interface da rede de trabalho com *sockets* TCP/IP.

A interface do sistema para os processos clientes é equivalente à do sistema de arquivos do Unix, sendo que os operadores dessa interface não são executados no nó do cliente, mas empacotados em mensagens enviadas ao nó do servidor de arquivos. Esses operadores interagem com a interface da rede de trabalho que realiza as trocas de mensagens entre os processos clientes e o processo servidor de arquivos. O servidor de arquivos interage diretamente com a interface da rede de trabalho e usa a interface de programação local do Linux para atender as requisições dos clientes.

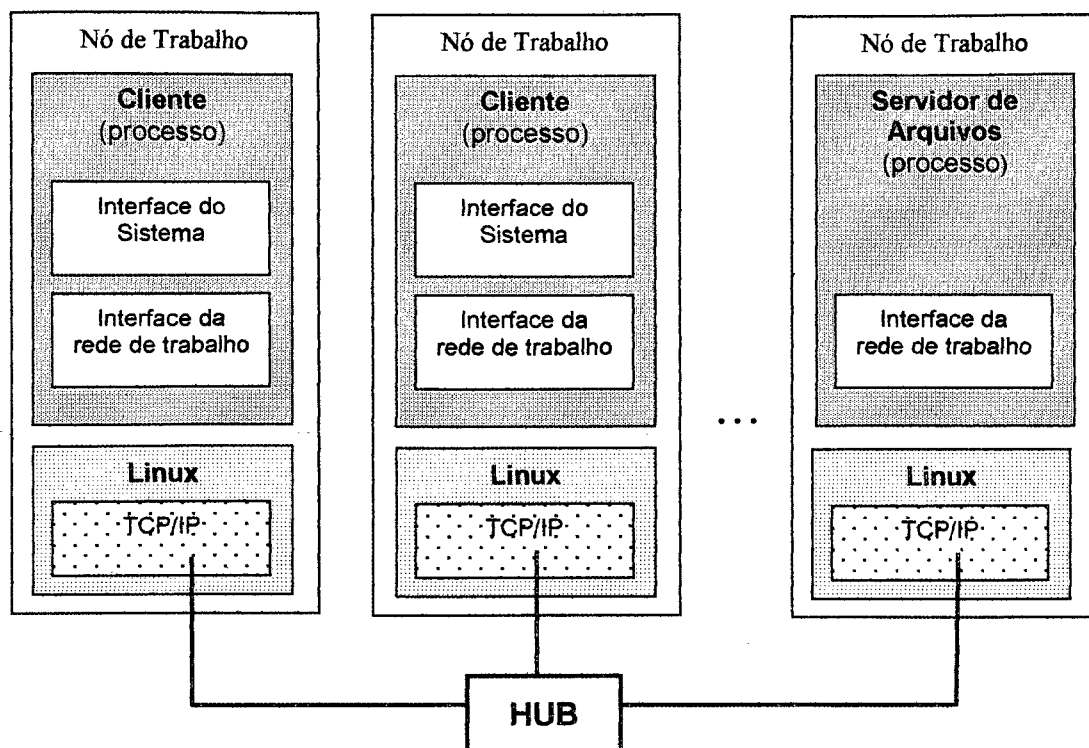


Fig 5.2 – Servidor de Arquivos.

O servidor de arquivos do cluster é implementado com processos regulares do Linux, executando em um único nó de trabalho.

O servidor de arquivos é baseado no modelo cliente-servidor. Deste 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.

5.2.1 Interface do Sistema

A interface do sistema para os processos clientes é implementada pela biblioteca *libcsa* que substitui parcialmente a biblioteca *libc* original do Linux. Os operadores implementados na *libcsa* são os seguintes: *open*, *close*, *read*, *write*, *link*, *unlink*, *symlink*, *truncate*, *lseek*, *ftruncate*, *chdir*, *rename*, *mkdir*, *rmdir*, *mknod*, *chown*, *dup*, *creat*, *access*, *stat*, *chmod*, *utime*. Os operadores que não se referem ao sistema de arquivos são executados localmente a partir da biblioteca *libc* original do Linux.

A biblioteca *libcsa*, deve ser priorizada no momento da compilação do programa cliente. Com isso, a biblioteca *libcsa* atenderá as chamadas de sistema que foram implementadas pelo servidor de arquivos, e a biblioteca *libc* atenderá localmente as demais chamadas de sistema.

Para a execução das chamadas de sistema, os operadores da biblioteca *libcsa* interagem com o processo servidor de arquivos através de troca de mensagens. Para cada chamada de sistema, é montada uma mensagem contendo a identificação da chamada de sistema e seus argumentos.

5.2.2 Interface da Rede de Trabalho

A interface da rede de trabalho está implementada provisoriamente com *sockets* TCP/IP. Esta interface será futuramente substituída por outra mais adequada ao cluster alvo que está sendo definida em outras dissertações [REC02, BOG02].

Os operadores que compõem esta interface são definidos em [COR99] e mostrados a seguir.

Send (proc, msg, length);

Esse operador é usado por um processo para enviar ao processo *proc* a mensagem *msg* de tamanho *length*. O código que implementa este operador é mostrado na figura 5.3.

```
1 void Send(int proc, const message *msg, int length)
2 {
3     n_bytes = send(proc, msg, length, 0);
4     if (n_bytes != length){
5         fprintf(stderr, "Erro em send");
6         exit(0);
7     }
8 }
```

Fig. 5.3 – Operador *Send*.

A chamada de sistema *send* do protocolo TCP/IP, na linha 3 da figura 5.3, é usada tanto pelos clientes quanto pelo servidor de arquivos para envio de mensagens. O argumento *proc* especifica o *socket* que será usado, o argumento *msg* fornece o endereço de memória dos dados a serem enviados, o argumento *length* especifica o número de bytes a serem enviados e o último argumento, *flags*, é sempre igual a 0 para atender a semântica bloqueante do operador *Send*.

Receive (proc, msg, length);

Esse operador é usado por um processo para receber do processo *proc* a mensagem *msg* de tamanho *length*. O código que implementa este operador é mostrado na figura 5.4.

```
1 void Receive(int proc, message *msg, int length)
2 {
3     n_bytes = recv(proc, msg, TAM_MSG, 0);
4     if (n_bytes < 0) {
5         perror("Erro em recv");
6         exit(0);
7     }
8 }
```

Fig. 5.4 – Operador *Receive*.

A chamada de sistema *recv* do protocolo TCP/IP, na linha 3 da figura 5.4, é usada tanto pelos clientes quanto pelo servidor de arquivos para receber mensagens. O argumento *proc* especifica o *socket* que será usado, o argumento *msg* especifica o endereço de memória no qual os dados devem ser colocados, o argumento *TAM_MSG* especifica o comprimento do *buffer* e o último argumento, *flags*, é sempre igual a 0 para atender a semântica bloqueante do operador *Receive*.

ReceiveAny (*proc*, *msg*, *length*);

Esse operador é usado por um processo para receber de um processo qualquer a mensagem *msg* de tamanho *length*. O código que implementa este operador é mostrado na figura 5.5.

```

1 void ReceiveAny(int proc, message *msg, int length)
2 {
3     n_bytes = recv(proc, msg, TAM_MSG, 0);
4
5     if (n_bytes < 0) {
6         perror("Erro em recv ");
7         exit(0);
8     }
9 }

```

Fig. 5.5 – Operador *ReceiveAny*.

A chamada de sistema *recv* do protocolo TCP/IP, na linha 3 da figura 5.5, é usada exclusivamente pelo servidor de arquivos para receber mensagens dos processos clientes. Os argumentos dessa chamada de sistema são os mesmos definidos para a chamada de sistema *recv* no operador *Receive*, descrito anteriormente.

Esses três operadores são suficientes para as comunicações do modelo cliente-servidor. Assim, um cliente executa o operador *Send* (para o envio de uma requisição de serviço) seguido de *Receive* (para a recepção da resposta). O servidor de arquivos executa o operador *ReceiveAny* (para a recepção de uma requisição de serviços) e *Send* (para o envio da resposta).

5.3 Detalhes de Implementação

Este tópico descreve as particularidades do desenvolvimento do servidor de arquivos proposto neste trabalho. No subtópico 5.3.1 é descrita a comunicação cliente-servidor usada nesta implementação. No subtópico 5.3.2 é apresentada a estrutura da mensagem usada na comunicação entre os processos clientes e o processo servidor. A

seguir, nos subtópicos 5.3.3 e 5.3.4, são brevemente descritos os processos cliente e servidor, respectivamente.

5.3.1 Comunicação Cliente-Servidor

Para a criação do servidor de arquivos foi utilizada a técnica de servidores concorrentes com *sockets* TCP/IP [STE98], conforme descrito em 3.6.

Na inicialização do servidor é realizada a criação de um *socket* TCP/IP, com estabelecimento de um endereço local para o *socket* 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. Quando uma solicitação de conexão de um processo cliente chega, o sistema operacional cria um novo *socket* que tem seu destino conectado a esse cliente específico. Em seguida, o servidor executa um *fork* para a criação de um processo servidor filho.

O processo servidor filho fecha o *socket* original e, usando o novo *socket*, executa um outro laço de repetição infinito, através do qual passa a trocar mensagens com esse cliente específico. O processo servidor pai fecha o novo *socket* e permanece com o *socket* original aberto, recebendo novas solicitações de outros clientes.

Na inicialização do cliente, um novo *socket* TCP/IP é criado, o endereço local do *socket* é definido e a requisição de conexão ao servidor é realizada. Se não houverem erros na execução da requisição de conexão, o novo *socket* é retornado ao processo cliente que pode então iniciar a troca de mensagens com o processo servidor filho. Senão, o erro associado à requisição de conexão é retornado ao processo cliente e este termina a execução.

5.3.2 Estrutura da mensagem

A mensagem trocada entre os processos clientes e o processo servidor é formada 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.

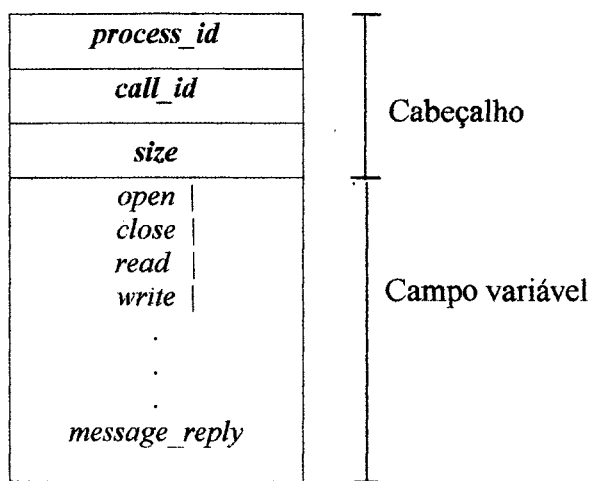


Fig. 5.6 – Formato da mensagem.

Os itens da mensagem têm os seguintes significados:

- *process_id* = identificação do processo cliente;
- *call_id* = identificação da chamada de sistema;
- *size* = tamanho da mensagem.

Os itens *process_id* e *size* não são utilizados nesta implementação, mas são mantidos no cabeçalho da mensagem por serem necessários à outros trabalhos [BOG02, REC02].

A declaração na linguagem C correspondente à mensagem é apresentada na figura 5.7.

```
typedef struct {
    header headr;
    union {
        s_open          sopen;
        s_close         sclose;
        s_creat         screat;
        s_read          sread;
        s_write         swrite;
        .
        .
        .
        message_reply  m_reply;
    } u_call;
} message;
```

Fig. 5.7 – Estrutura da mensagem definida na linguagem C.

O campo variável das chamadas de sistema é uma estrutura definida de acordo com os argumentos de cada chamada. No caso específico da chamada de sistema *write*, além dos argumentos a estrutura possui um *buffer* para conter os dados a serem escritos. A figura 5.8 mostra, como exemplo, a definição da mensagem correspondente à chamada de sistema *open*.

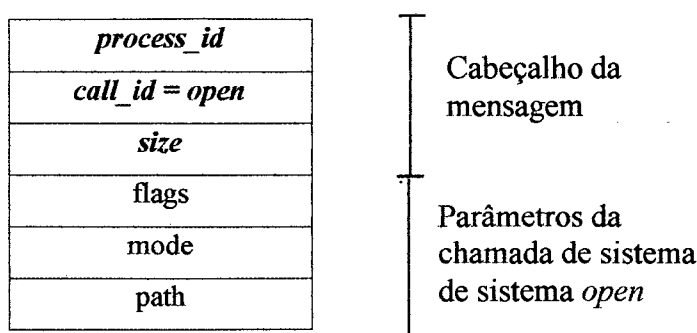


Fig. 5.8 – Mensagem correspondente à chamada de sistema *open*.

A declaração na linguagem C do campo variável correspondente à chamada de sistema *open* é mostrada na figura. 5.9.

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

Fig. 5.9 – Chamada de sistema *open* definida na linguagem C.

Como parte do campo variável, existe ainda uma estrutura que corresponde às mensagens de retorno que são enviadas do processo servidor de arquivos aos processos clientes. Essa estrutura contém o resultado da execução, no processo servidor, da chamada de sistema solicitada. Em caso de erro na execução das chamadas de sistema, o valor da variável global *errno* também é colocada nessa estrutura. Essa estrutura possui também um *buffer* para conter os dados lidos na chamada de sistema *read*, e estruturas específicas utilizadas nas chamadas de sistema *stat* e *utime*. A

declaração na linguagem C do campo variável correspondente à essa estrutura é mostrada na figura. 5.10.

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

Fig. 5.10 – Mensagem de retorno definida na linguagem C.

5.3.3 Processos Clientes

Os processos clientes utilizam, para a maioria das chamadas de sistema relacionadas ao sistema de arquivos, a biblioteca *libcsa*, conforme exposto em 5.2.1. Esta biblioteca implementa grande parte das chamadas de sistema relativas ao sistema de arquivos do Linux.

O algoritmo genérico das funções da biblioteca *libcsa* é mostrado na figura 5.11. Nesse algoritmo, é realizada inicialmente uma verificação dos possíveis erros relacionados aos argumentos utilizados, pelo processo cliente, na chamada de sistema (linha 1). Se algum argumento inválido for encontrado, a variável global *errno* é atualizada e o resultado é retornado imediatamente (linhas 2-4). Senão, a mensagem é montada e enviada ao servidor (linhas 5-7). A execução do cliente fica então suspensa pela espera da mensagem de retorno do processo servidor (linha 8). Na chegada da mensagem de retorno, se houver erro de execução, a variável global *errno* é atualizada (linhas 9-11). Em seguida, o resultado é retornado ao cliente (linha 12).

```

1  se existirem erros nos argumentos;
2
3  então
4      atualiza erro;
5      retorna com erro;
6
7  senão
8      monta a mensagem;
9      envia a mensagem ao servidor;
10     espera a mensagem de retorno do servidor;
11     se existirem erros de execução;
12
13     então
14         atualiza erro;
15
16     retorna com resultado;

```

Fig. 5.11 – Algoritmo genérico das funções da biblioteca *libcsa*.

5.3.4 Servidor de Arquivos

O algoritmo genérico do servidor de arquivos é mostrado na figura 5.12. Esse algoritmo procede à inicialização do servidor, conforme descrito em 5.3.1 (linha 1). A seguir, um laço de repetição é executado, no qual o processo servidor fica a espera de pedidos de conexões dos processos clientes. Quando chega um pedido de conexão, o processo servidor cria um processo servidor filho para atender esse cliente específico (linhas 2-6). Então, o processo servidor filho executa um outro laço de repetição no qual fica a espera de requisições desse cliente (linhas 7-8). Quando uma requisição chega, a chamada de sistema identificada no cabeçalho da mensagem é usada para desviar a execução do programa para o código associado a ela (linha 9). A chamada de sistema é então executada e, em seguida, são realizadas algumas verificações relacionadas ao valor de retorno da chamada de sistema (linhas 10-12).

A mensagem de retorno ao processo cliente é montada e enviada (linhas 13-14).

```

1  inicializa o servidor de arquivos;
2  repita
3      se chegar pedido de conexão de um processo cliente
4      então
5          cria um processo servidor filho;
6  fim;
7      repita (processo servidor filho)
8          espera mensagem do cliente;
9          caso chamada de sistema;
           .
           .
           .
10         open:
11             executa a chamada de sistema;
12             verifica resultado da execução;
13             monta a mensagem de retorno;
14             envia a mensagem de retorno ao cliente;
15         fim;

```

Fig. 5.12 – Algoritmo genérico do servidor de arquivos.

5.4 Exemplo

Para melhor entender o funcionamento do servidor de arquivos, um exemplo é apresentado a seguir mostrando todas as etapas envolvidas na execução de uma chamada de sistema.

A figura 5.13 mostra o código da chamada de sistema *open* executada pelo cliente.

```

1 int open(const char *name, int flags, ...)
2 {
3     va_list ap;
4     int tam_msg, aux;
5
6     if (strlen(name) > _POSIX_PATH_MAX) {
7         errno = ENAMETOOLONG;
8         return(-1);
9     }
10    primeira_execucao();
11    va_start(ap, flags);
12    strcpy(m.u_call.sopen.path, name);
13    m.u_call.sopen.flags = flags;
14    m.headr.call = 5;
15    m.headr.id = 1;
16    aux = va_arg(ap, int);
17    m.u_call.sopen.mode = aux;
18    va_end(ap);
19    tam_msg =  strlen(m.u_call.sopen.path)+
20                sizeof(m.u_call.sopen.flags)+
21                sizeof(m.u_call.sopen.mode)+
22                sizeof(m.headr);
23    Send(sock_fd, &m, tam_msg);
24    tam_msg = sizeof(m.u_call.m_reply.result)+
25                sizeof(m.u_call.m_reply.m_errno)+
26                sizeof(m.headr);
27    Receive(sock_fd, &m, tam_msg);
28    if (m.u_call.m_reply.result == -1)
29        errno = m.u_call.m_reply.m_errno;
30    return (m.u_call.m_reply.result);
31 }

```

Fig. 5.13 - Código do processo cliente.

Linhas 1-9 A chamada de sistema *open* faz parte do conjunto de chamadas de sistema implementadas na biblioteca *libcsa*. A chamada de sistema *open* executada pelo cliente verifica se o tamanho do nome do arquivo

solicitado é maior que o tamanho máximo admitido pelo núcleo do Linux. Se for, a variável global *errno* é atualizada com o valor *ENAMETOOLONG* e a execução é retornada ao cliente.

Linha 10 A função *primeira_execucao* verifica se é a primeira interação desse cliente com o servidor de arquivos. Se for, um novo *socket* é criado e a conexão com o servidor é estabelecida. Senão, é utilizado o *socket* que já tenha sido definido anteriormente por outra interação do processo cliente com o servidor de arquivos. A função *primeira_execucao* é detalhada mais adiante.

Linhas 11-18 A mensagem que será enviada ao servidor é montada. Os argumentos da chamada de sistema são copiados para a estrutura definida para *open*.

Linhas 19-27 O tamanho da mensagem é calculado e o envio da mensagem é realizado pelo operador *Send*. O tamanho da mensagem de retorno é calculado e o processo cliente fica suspenso, esperando a mensagem de retorno pelo operador *Receive*.

Linhas 28-31 Se o valor retornado indicar erro de execução, a variável global *errno* é atualizada. Este valor é retornado ao cliente pela execução do operador *return*.

A figura 5.14 mostra o código em C da função *primeira_execucao*.

```
1 void primeira_execucao(void)
2 {
3     if (flag == 0)
4     {
5         flag = 1;
6         sock_fd = inicializa_cliente();
7     }
8 }
```

Fig. 5.14 – Código da função *primeira_execucao*.

Linhas 1-8 A função *primeira_execucao* utiliza uma variável global *flag* para testar se este cliente está interagindo com o processo servidor de arquivos pela primeira vez. Caso esteja, a função *inicializa_cliente* é chamada e deve retornar um novo *socket* para que o processo cliente inicie sua interação com o processo servidor de arquivos.

A figura 5.15 mostra o código da função *inicializa_cliente*.

```

1  int inicializa_cliente(void)
2  {
3      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
4          perror(" Erro na criação do socket");
5          exit(0);
6      }
7
7      bzero(&cli_addr, sizeof(cli_addr));
8      cli_addr.sin_family = AF_INET;
9      cli_addr.sin_port = htons(SERV_PORT);
10     inet_pton(AF_INET, ender, &cli_addr.sin_addr);
11
11     if ((connect(sockfd, (SA *) &cli_addr,
12                 sizeof(cli_addr))) < 0) {
13         printf("\nNo inicializa_cliente:\n");
14         perror("Erro em connect");
15         exit(0);
16     }
17
17     return (sockfd);
18 }

```

Fig. 5.15 – Código da função *inicializa_cliente*.

Linhas 1-6 Na função *inicializa_cliente*, 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 cliente encerra a sua execução. Senão, um *socket* TCP é criado, o qual será usado para as interações com o processo servidor.

Linhas 7-10 A estrutura de endereçamento do *socket* é utilizada pela maioria das funções *sockets*, e deve ser preenchida com os valores adequados para o protocolo TCP.

Linhas 11-16 A função *connect* é utilizada para estabelecer uma conexão com o processo servidor de arquivos. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o processo encerra sua execução.

Linhas 17-18 A função *inicializa_cliente* encerra sua execução retornando o *socket* criado para a função correspondente à chamada de sistema *open*.

A figura 5.16 mostra o código executado pelo servidor de arquivos para o atendimento dessa chamada de sistema *open*.

```

1  main()
2  {
3      inicializa_servidor();

4      for ( ; ; ) {
5          clilen = sizeof(cliaddr);
6          if ((connfd = accept(listenfd, (SA *) &cliaddr,
7                          &clilen)) < 0) {

8              perror("Erro em accept");
9              exit(0);
10         }

11         if ( (childpid = fork()) == 0) {
12             close(listenfd);
13             while (1) {
14                 bzero(&mm, sizeof(mm));
15                 ReceiveAny(connfd, &mm, lenght);
16                 switch (mm.headr.call) {

17                 case 1:
                     .
                     .
                     .

```

Continua na próxima página

```

18         case 5: // open
19             {
20                 fd = open(mm.u_call.sopen.path,
21                         mm.u_call.sopen.flags,
22                         mm.u_call.sopen.mode);
23
24                 mm.u_call.m_reply.result = fd;
25
26                 if (fd == -1)
27                     mm.u_call.m_reply.m_errno = errno;
28
29                 tam_msg = sizeof(mm.u_call.m_reply.result) +
30                          sizeof(mm.u_call.m_reply.m_errno) +
31                          sizeof(mm.headr);
32
33                 Send(connfd, &mm, tam_msg);
34
35                 break;
36             }
37         case 6:
38             .
39             .
40             .
41             .
42             .
43             .
44             .
45             .
46             .
47             .
48             .
49             .
50             .
51             .
52             .
53             .
54             .
55             .
56             .
57             .
58             .
59             .
60             .
61             .
62             .
63             .
64             .
65             .
66             .
67             .
68             .
69             .
70             .
71             .
72             .
73             .
74             .
75             .
76             .
77             .
78             .
79             .
80             .
81             .
82             .
83             .
84             .
85             .
86             .
87             .
88             .
89             .
90             .
91             .
92             .
93             .
94             .
95             .
96             .
97             .
98             .
99             .
100            .
101            .
102            .
103            .
104            .
105            .
106            .
107            .
108            .
109            .
110            .
111            .
112            .
113            .
114            .
115            .
116            .
117            .
118            .
119            .
120            .
121            .
122            .
123            .
124            .
125            .
126            .
127            .
128            .
129            .
130            .
131            .
132            .
133            .
134            .
135            .
136            .
137            .
138            .
139            .
140            .
141            .
142            .
143            .
144            .
145            .
146            .
147            .
148            .
149            .
150            .
151            .
152            .
153            .
154            .
155            .
156            .
157            .
158            .
159            .
160            .
161            .
162            .
163            .
164            .
165            .
166            .
167            .
168            .
169            .
170            .
171            .
172            .
173            .
174            .
175            .
176            .
177            .
178            .
179            .
180            .
181            .
182            .
183            .
184            .
185            .
186            .
187            .
188            .
189            .
190            .
191            .
192            .
193            .
194            .
195            .
196            .
197            .
198            .
199            .
200            .
201            .
202            .
203            .
204            .
205            .
206            .
207            .
208            .
209            .
210            .
211            .
212            .
213            .
214            .
215            .
216            .
217            .
218            .
219            .
220            .
221            .
222            .
223            .
224            .
225            .
226            .
227            .
228            .
229            .
230            .
231            .
232            .
233            .
234            .
235            .
236            .
237            .
238            .
239            .
240            .
241            .
242            .
243            .
244            .
245            .
246            .
247            .
248            .
249            .
250            .
251            .
252            .
253            .
254            .
255            .
256            .
257            .
258            .
259            .
260            .
261            .
262            .
263            .
264            .
265            .
266            .
267            .
268            .
269            .
270            .
271            .
272            .
273            .
274            .
275            .
276            .
277            .
278            .
279            .
280            .
281            .
282            .
283            .
284            .
285            .
286            .
287            .
288            .
289            .
290            .
291            .
292            .
293            .
294            .
295            .
296            .
297            .
298            .
299            .
300            .
301            .
302            .
303            .
304            .
305            .
306            .
307            .
308            .
309            .
310            .
311            .
312            .
313            .
314            .
315            .
316            .
317            .
318            .
319            .
320            .
321            .
322            .
323            .
324            .
325            .
326            .
327            .
328            .
329            .
330            .
331            .
332            .
333            .
334            .
335            .
336            .
337            .
338            .
339            .
340            .
341            .
342            .
343            .
344            .
345            .
346            .
347            .
348            .
349            .
350            .
351            .
352            .
353            .
354            .
355            .
356            .
357            .
358            .
359            .
360            .
361            .
362            .
363            .
364            .
365            .
366            .
367            .
368            .
369            .
370            .
371            .
372            .
373            .
374            .
375            .
376            .
377            .
378            .
379            .
380            .
381            .
382            .
383            .
384            .
385            .
386            .
387            .
388            .
389            .
390            .
391            .
392            .
393            .
394            .
395            .
396            .
397            .
398            .
399            .
400            .
401            .
402            .
403            .
404            .
405            .
406            .
407            .
408            .
409            .
410            .
411            .
412            .
413            .
414            .
415            .
416            .
417            .
418            .
419            .
420            .
421            .
422            .
423            .
424            .
425            .
426            .
427            .
428            .
429            .
430            .
431            .
432            .
433            .
434            .
435            .
436            .
437            .
438            .
439            .
440            .
441            .
442            .
443            .
444            .
445            .
446            .
447            .
448            .
449            .
450            .
451            .
452            .
453            .
454            .
455            .
456            .
457            .
458            .
459            .
460            .
461            .
462            .
463            .
464            .
465            .
466            .
467            .
468            .
469            .
470            .
471            .
472            .
473            .
474            .
475            .
476            .
477            .
478            .
479            .
480            .
481            .
482            .
483            .
484            .
485            .
486            .
487            .
488            .
489            .
490            .
491            .
492            .
493            .
494            .
495            .
496            .
497            .
498            .
499            .
500            .
501            .
502            .
503            .
504            .
505            .
506            .
507            .
508            .
509            .
510            .
511            .
512            .
513            .
514            .
515            .
516            .
517            .
518            .
519            .
520            .
521            .
522            .
523            .
524            .
525            .
526            .
527            .
528            .
529            .
530            .
531            .
532            .
533            .
534            .
535            .
536            .
537            .
538            .
539            .
540            .
541            .
542            .
543            .
544            .
545            .
546            .
547            .
548            .
549            .
550            .
551            .
552            .
553            .
554            .
555            .
556            .
557            .
558            .
559            .
560            .
561            .
562            .
563            .
564            .
565            .
566            .
567            .
568            .
569            .
570            .
571            .
572            .
573            .
574            .
575            .
576            .
577            .
578            .
579            .
580            .
581            .
582            .
583            .
584            .
585            .
586            .
587            .
588            .
589            .
590            .
591            .
592            .
593            .
594            .
595            .
596            .
597            .
598            .
599            .
600            .
601            .
602            .
603            .
604            .
605            .
606            .
607            .
608            .
609            .
610            .
611            .
612            .
613            .
614            .
615            .
616            .
617            .
618            .
619            .
620            .
621            .
622            .
623            .
624            .
625            .
626            .
627            .
628            .
629            .
630            .
631            .
632            .
633            .
634            .
635            .
636            .
637            .
638            .
639            .
640            .
641            .
642            .
643            .
644            .
645            .
646            .
647            .
648            .
649            .
650            .
651            .
652            .
653            .
654            .
655            .
656            .
657            .
658            .
659            .
660            .
661            .
662            .
663            .
664            .
665            .
666            .
667            .
668            .
669            .
670            .
671            .
672            .
673            .
674            .
675            .
676            .
677            .
678            .
679            .
680            .
681            .
682            .
683            .
684            .
685            .
686            .
687            .
688            .
689            .
690            .
691            .
692            .
693            .
694            .
695            .
696            .
697            .
698            .
699            .
700            .
701            .
702            .
703            .
704            .
705            .
706            .
707            .
708            .
709            .
710            .
711            .
712            .
713            .
714            .
715            .
716            .
717            .
718            .
719            .
720            .
721            .
722            .
723            .
724            .
725            .
726            .
727            .
728            .
729            .
730            .
731            .
732            .
733            .
734            .
735            .
736            .
737            .
738            .
739            .
740            .
741            .
742            .
743            .
744            .
745            .
746            .
747            .
748            .
749            .
750            .
751            .
752            .
753            .
754            .
755            .
756            .
757            .
758            .
759            .
760            .
761            .
762            .
763            .
764            .
765            .
766            .
767            .
768            .
769            .
770            .
771            .
772            .
773            .
774            .
775            .
776            .
777            .
778            .
779            .
780            .
781            .
782            .
783            .
784            .
785            .
786            .
787            .
788            .
789            .
790            .
791            .
792            .
793            .
794            .
795            .
796            .
797            .
798            .
799            .
800            .
801            .
802            .
803            .
804            .
805            .
806            .
807            .
808            .
809            .
810            .
811            .
812            .
813            .
814            .
815            .
816            .
817            .
818            .
819            .
820            .
821            .
822            .
823            .
824            .
825            .
826            .
827            .
828            .
829            .
830            .
831            .
832            .
833            .
834            .
835            .
836            .
837            .
838            .
839            .
840            .
841            .
842            .
843            .
844            .
845            .
846            .
847            .
848            .
849            .
850            .
851            .
852            .
853            .
854            .
855            .
856            .
857            .
858            .
859            .
860            .
861            .
862            .
863            .
864            .
865            .
866            .
867            .
868            .
869            .
870            .
871            .
872            .
873            .
874            .
875            .
876            .
877            .
878            .
879            .
880            .
881            .
882            .
883            .
884            .
885            .
886            .
887            .
888            .
889            .
890            .
891            .
892            .
893            .
894            .
895            .
896            .
897            .
898            .
899            .
900            .
901            .
902            .
903            .
904            .
905            .
906            .
907            .
908            .
909            .
910            .
911            .
912            .
913            .
914            .
915            .
916            .
917            .
918            .
919            .
920            .
921            .
922            .
923            .
924            .
925            .
926            .
927            .
928            .
929            .
930            .
931            .
932            .
933            .
934            .
935            .
936            .
937            .
938            .
939            .
940            .
941            .
942            .
943            .
944            .
945            .
946            .
947            .
948            .
949            .
950            .
951            .
952            .
953            .
954            .
955            .
956            .
957            .
958            .
959            .
960            .
961            .
962            .
963            .
964            .
965            .
966            .
967            .
968            .
969            .
970            .
971            .
972            .
973            .
974            .
975            .
976            .
977            .
978            .
979            .
980            .
981            .
982            .
983            .
984            .
985            .
986            .
987            .
988            .
989            .
990            .
991            .
992            .
993            .
994            .
995            .
996            .
997            .
998            .
999            .
1000           .

```

Fig. 5.16 – Código do servidor de arquivos.

- Linhas 1-3 O processo servidor de arquivos executa a função de inicialização *inicializa_servidor*, na qual um *socket* é criado e o endereço e porta local são definidos. A função *inicializa_servidor* é detalhada mais adiante.
- Linhas 4-10 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*. 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*. O processo servidor mantém, então, dois *sockets* abertos, o original e o novo.

- Linhas 11-12 O processo servidor de arquivos executa um *fork*, criando um processo servidor filho e retorna para o laço de repetição. O processo servidor filho fecha o *socket* original através da função *close* e atende as requisições do cliente através do novo *socket*.
- Linhas 13-16 O processo servidor filho executa um outro laço de repetição infinito, no qual fica à espera de mensagens através do operador *ReceiveAny*. Na chegada de uma nova mensagem, o operador *switch* da linguagem C verifica que a chamada de sistema solicitada é *open*, e desvia a execução do programa para o código correspondente.
- Linhas 18-25 O processo servidor filho interage com o sistema de arquivos do Linux e executa a chamada de sistema *open*. O retorno da chamada de sistema é colocado na estrutura da mensagem de retorno. Se o valor retornado indicar erro de execução, o erro associado a esse valor (indicado pela variável global *errno*) também é colocado na mensagem de retorno.
- Linhas 26-31 O tamanho da mensagem de retorno é calculado e a mensagem é enviada ao processo cliente pelo operador *Send*. Em seguida, o operador *break* da linguagem C é executado fazendo com que o processo servidor filho retorne ao laço de repetição infinito.
- Linha 36 O processo servidor pai fecha o novo *socket* e retorna ao laço de repetição infinito.

A figura 5.17 mostra o código da função *inicializa_servidor*.

```

1  inicializa_servidor(void)
2  {
3      if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
4          perror("Erro na criação do socket");
5          exit(0);
6      }

7      bzero(&servaddr, sizeof(servaddr));
8      servaddr.sin_family      = AF_INET;
9      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
10     servaddr.sin_port        = htons(SERV_PORT);

11     if (bind(listenfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
12     {
13         perror("Erro na execução do bind");
14         exit(0);
15     }

16     if ((listen(listenfd, LISTENQ)) < 0) {
17         perror("Erro em listen");
18         exit(0);
19     }
20 }

```

Fig. 5.17 - Código da função *inicializa_servidor*.

- Linhas 1-6 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 arquivos encerra a sua execução. Senão, um *socket* TCP é criado, o qual será usado para as interações com os processos clientes.
- Linhas 7-10 A estrutura de endereçamento do *socket* deve ser preenchida com os valores adequados para o protocolo TCP.
- Linhas 11-15 A função *bind* é utilizada pelo processo servidor de arquivos 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 arquivos encerra a sua execução.

Linhas 16-20 A função *listen* é utilizada pelo processo servidor de arquivos para preparar o *socket* para aceitar as conexões com os processos clientes. Se o valor retornado indicar erro de execução, uma mensagem de erro é impressa e o processo servidor de arquivos encerra a sua execução.

6. Conclusão

Neste capítulo são apresentadas as contribuições deste trabalho e projetos futuros. No tópico 6.1 são apresentados as contribuições e no tópico 6.2 são descritos os projetos futuros.

6.1 Contribuições

Este texto apresentou o projeto e a implementação de um servidor de arquivos para um cluster de computadores derivado do multicomputador Crux. Utilizou-se o modelo cliente-servidor para as comunicações entre processos clientes e o processo servidor de arquivos. Os processos clientes interagem com o processo servidor de arquivos através de *sockets* TCP pela rede de trabalho, com um mecanismo específico de troca de mensagens desenvolvido para esse cluster.

O processo servidor interage com o sistema de arquivos do Linux para a execução das chamadas de sistema solicitadas por processos clientes e com a rede de trabalho. A rede de trabalho é responsável pelo trânsito de mensagens entre processos clientes e o processo servidor de arquivos. Os processos clientes possuem uma interface própria equivalente à do sistema Unix, sendo que os operadores não são processados localmente e sim enviados ao processo servidor através da rede de trabalho.

Foi desenvolvida a biblioteca *libcsa* para o servidor de arquivos, a qual contém funções correspondentes às chamadas de sistema. Esta biblioteca deve ser priorizada na compilação do programa cliente. A maioria das chamadas de sistema relativas ao sistema de arquivos do Linux são atendidas pela biblioteca *libcsa*.

Para a implementação do servidor de arquivos foi utilizada a técnica de servidores concorrentes [STE97], a qual cria um processo servidor filho para atender cada novo processo cliente. Esta técnica torna o processo servidor mais eficiente na medida em que cada processo cliente é atendido imediatamente por um processo servidor filho.

Este trabalho produziu um componente essencial para o cluster alvo.

6.2 Perspectivas Futuras

Para o desenvolvimento deste trabalho, a interface da rede de trabalho foi implementada provisoriamente com *sockets* TCP/IP. Em breve, essa interface deverá ser substituída por outra, objeto de pesquisa de outras dissertações ([BOG02], [REC02]).

O ambiente de execução de programas paralelos do cluster alvo prevê que o servidor de arquivos seja o único elemento contido em um nó de trabalho. Assim, em uma etapa mais adiantada do desenvolvimento de software para esse cluster, o sistema de arquivos poderá ser implementado como uma entidade autônoma, prescindindo do suporte de um sistema operacional completo.

7. Referências Bibliográficas

- BOG02 Bogo, Madianita. *Interface da Rede de Controle do Cluster Clux*. Dissertação de Mestrado, CPGCC / UFSC. 2002.
- CIA99 Ciaccio, Giuseppe. *A Communication System for Efficient Parallel Processing on Clusters of Personal Computer*. Genova, 1999. Disponível em: <<http://www.disi.unige.it/project/gamma/>>. Acesso em: 09 outubro 2001.
- COM98 Comer, D. E. *Interligação em Rede com TCP/IP*. Campus, 1998.
- COR99 Corso, T. B. *Crux: Ambiente Multicomputador Reconfigurável por Demanda*, Tese de Doutorado, CPGEE / UFSC, 1999.
- DIE01 Dietz, H. *Linux Parallel Processing Using Clusters*. Disponível em: <<http://aggregate.org/pplinux/19960315/ppcluster.html>>. Acesso em: 20 julho 2001.
- FER01 Ferreto, T. C, et al. *CPAD-PUCRS/HP : GNU/Linux como Plataforma para Pesquisa em Alto Desempenho*. Disponível em: <<http://www.inf.pucrs.br/~derose/files/wsl2001.pdf>>. Acesso em: 20 outubro 2001.
- MAC97 Machado, F. B., Maia, L. P. *Arquitetura de Sistemas Operacionais*, LTC, 1977.
- MAR01 Martins, S. L.; Ribeiro, C.C.C.; Rodriguez, N. L. R. *Ferramentas para Programação Paralela em Ambientes de Memória Distribuída*. Rio de Janeiro, 1996. Acesso em: 10 novembro 2001. Disponível em: <<http://www.inf.puc-rio.br/~celso/publicacoes.html>>.

- MER96 Merkle, C. *Ambiente para Execução de Programas Paralelos Escritos na Linguagem Superpascal em um Multicomputador com Rede de Interconexão Dinâmica*, Dissertação de Mestrado, CPGCC / UFSC, 1996.
- MPI01 *MPI (Message Passing Interface)*. Acesso em: 15 setembro 2001. Disponível em <<http://www.netlib.org/pvm3/book/node15.html>>.
- MYR01 *Myrinet Overview*. Acesso em: 15 setembro 2001. Disponível em: <<http://www.myri.com/myrinet/overview/>>.
- NOR96 Northrup, C. *Programming with Unix Threads*. John Wiley & Sons, 1996.
- OLI00 Oliveira, R. S; Carissimi, A. S; Toscani, S. S. *Sistemas Operacionais*. Sagra Luzzatto, 2000.
- PVM01 *PVM (Parallel Virtual Machine)*. Acesso em: 16 setembro 2001. Disponível em <<http://shiva.di.uminho.pt/~pina/textos/pvm/node1.html>>.
- QUA01 Quadros, M. G. *Tecnologias, Protocolos e Aplicações em Redes de Alta Velocidade – Caracterização, Limitações e Campos em Aberto*. Disponível em: <<http://www.criticalsoftware.com/research/pdf/rt1.pdf>>. Acesso em: 10 outubro 2001.
- REC02 Rech, Luciana de Oliveira. *Interface da Rede de Trabalho do Cluster Clux*. Dissertação de Mestrado, CPGCC / UFSC. 2002.
- RIB01 Ribeiro, C.C.C; Rodriguez, N. L. R. *Otimização e Processamento Paralelo de Alto Desempenho*. Revista Puc Ciência, Rio de Janeiro, 1999. 45-48. Acesso em: 30 setembro 2001. Disponível em: <<http://www.inf.puc-rio.br/~celso/publicacoes.html>>.

- SCH01 Schweitzer, M. A.; Zumbusch, G.; Griebel, M. *Parnass2: A cluster of Dual Processor PCs*. Disponível em: <<http://wissrech.iam.uni-born.de/research/projects/parnass2>>. Acesso em: 20 julho 2001.
- SOA95 Soares, L. F. G., Lemos, G., Colcher, S. *Redes de Computadores: das LANs, MANs e WANs às redes ATM*, Campus, 1995.
- STE97 Stevens, W. R. *Unix Networking Programming*, Prentice Hall, 1997.
- TAN92 Tanenbaum, A. S. *Sistemas Operacionais Modernos*, Prentice Hall, 1992.
- TAN97 Tanenbaum, A. S. *Redes de Computadores*, Campus, 1997.
- TAN00 Tanenbaum, A. S., Woodhull, A. S. *Sistemas Operacionais: projeto e implementação*, 2. ed. – Bookman, 2000.
- TEI96 Teixeira, J. H. T., Sauv e, J. P., Moura, J. A. B. *Do mainframe para a computa o distribu da: simplificando a transi o*, Infobook, 1996.
- TEX01 *Texas Tech Tornado Cluster: A Linux/MPI Cluster For Parallel Programming Education And Research*. Disponível em <<http://www.acm.org/crossroads/xrds6-1/tornado.html>>. Acesso em: 23 julho 2001.
- THE01 *The PVM System*. Aceso em: 15 setembro 2001. Disponível em <<http://www.netlib.org/pvm3/book/node17.html>>.
- TSU01 Tsugawa, M. O. *Aspectos de Projeto de Sistemas de Processamento Paralelo Utilizando Tecnologias Emergentes de Comunica o*. S o Paulo, 2001. Acesso em: 24 agosto 2001. Disponível em: <<http://www.lsi.usp.br/~tsugawa/diss/motdiss2k1.pdf>>.