


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

Cláudia Heusi Silveira

 **GRUX : UM MECANISMO DE COMUNICAÇÃO
EM GRUPO PARA O AMBIENTE
PARALELO/DISTRIBUÍDO CRUX**

Dissertação submetida à Universidade Federal de Santa Catarina para a obtenção do grau de Mestre em Ciências da Computação

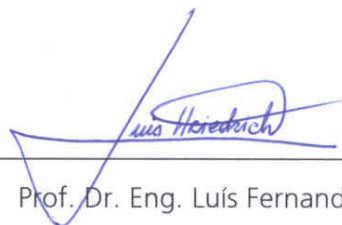
Luís Fernando Friedrich

Florianópolis, Dezembro de 2000

CRUX : UM MECANISMO DE COMUNICAÇÃO EM GRUPO PARA O AMBIENTE PARALELO/DISTRIBUÍDO CRUX

Cláudia Heusi Silveira

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciências da Computação área de concentração Sistemas de Computação, aprovada em sua forma final pelo Programa de Pós-Graduação em Ciências da Computação.



Prof. Dr. Eng. Luís Fernando Friedrich
Orientador



Prof. Dr. Eng. Fernando A. Gauthier
Coordenador do Programa

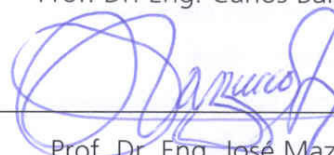
BANCA EXAMINADORA




Prof. Dr. Eng. Thadeu Botteri Corso



Prof. Dr. Eng. Carlos Barros Montez



Prof. Dr. Eng. José Mazzuco Junior



Prof. Dr. Eng. Rômulo Silva de Oliveira

AGRADECIMENTOS

Registro aqui meus agradecimentos a todas as pessoas que de uma forma ou de outra contribuíram para a conclusão deste trabalho. Em especial, cito aqueles que me apoiaram quando eu estava a ponto de desistir e que sem eles este trabalho não existiria:

Sérgio “Peão” Macedo, Milton “Miltinho” Pereira, Henrique “Animal” Muhle, Joel M. Crichigno Filho, Tatiane “Tati” Goetten, Guilhermina “Mina” Salasário, Dayna M. Bortoluzzi, Márcia M. Giraldi, Alessandra “Ale” D’Aquino, por me ensinar a viver sempre de bem com vida e ao meu orientador Luiz Fernando Friedrich, por todas as dicas, críticas, orientações. Ao Prof. Dr. – Ing. Walter L. Weingaertner que sem o seu consentimento, não poderia ter usado o LMP como laboratório para o meu trabalho. À minha família, meu porto seguro, especialmente meus pais, Irio e Solange, sem a educação que me deram, não seria o que sou; e Nádia e Cátia, minhas irmãs, pelas palavras amigas nas horas difíceis. Aos colegas do LMP, pelas incansáveis discussões a respeito de novas tecnologias e pesquisas em Ciência e Tecnologia. Ao pessoal da FUNCITEC, pelo apoio na fase final do trabalho. Ao pessoal do pólo aquático, que sem uma “válvula de escape” jamais teria conseguido. Meu muito obrigada a você, leitor.

“Não basta ensinar ao homem uma especialidade,
porque se tornará assim uma máquina utilizável
e não uma personalidade.
É necessário que adquira um sentimento,
um senso prático daquilo
que vale a pena ser empreendido,
daquilo que é belo,
do que é moralmente correto”
Albert Einstein

ÍNDICE DE FIGURAS

Figura 2.1 – Taxonomia de sistemas paralelos distribuídos.....	4
Figura 2.2 – Sistema multiprocessador baseado em barramento.....	5
Figura 2.3 – Sistemas multiprocessadores chaveados dos tipos <i>crossbar</i> e rede omega.	5
Figura 2.4 – Sistema multicomputador baseado em barramento.....	6
Figura 2.5 – Sistemas multicomputadores comutados dos tipos grelha e hipercubo.....	6
Figura 2.6 – Estrutura da arquitetura <i>cluster</i>	8
Figura 2.7 – Modelo de comunicação via memória compartilhada.....	12
Figura 2.8 – Modelo de comunicação via troca de mensagens.....	13
Figura 2.9 – Modelo cliente/servidor (envio/resposta)	17
Figura 2.10 – Chamada remota de procedimento RPC [TANENBAUM (1992)]	17
Figura 3.1 – Tipos de ordenação de mensagens.....	20
Figura 3.2 – Tipos de endereçamento.....	21
Figura 3.3 – a) Comunicação ponto-a-ponto e b) comunicação um-para-muitos.....	22
Figura 3.4 – Comunicação ponto-a-ponto versus comunicação em grupo [MACHADO (1997)].....	23
Figura 3.5 – Sobreposição de grupos.....	24
Figura 3.6 – Grupos abertos ou fechados.....	25
Figura 3.7 – Grupos pares ou hierárquicos.....	25
Figura 3.8 – Arquitetura de <i>hardware</i> do AMOEBA.....	27
Figura 3.9 – Protocolo com seqüenciador do AMOEBA.....	28
Figura 3.10 – Estrutura do AMOEBA para suporte à comunicação em grupo.....	29
Figura 3.11 – Exemplo do vetor utilizado por CBCAST.....	31
Figura 4.1 – A arquitetura do CRUX.....	37
Figura 4.2 – Estrutura interna do nó de trabalho.....	37
Figura 4.3 – Camadas do micronúcleo do CRUX [MONTEZ (1995)].....	39
Figura 4.4 – Camadas do micronúcleo com comunicação em grupo.....	41
Figura 4.5 – Tabela de grupos do servidor SG.....	43

Figura 4.6 – Ordenação consistente através do servidor de grupo.....	44
Figura 4.7 – Funcionamento da primitiva GcCreate.....	46
Figura 4.8 – Funcionamento da primitiva GcDestroy.....	47
Figura 4.9 – Funcionamento da primitiva GcJoin.....	48
Figura 4.10 – Funcionamento da primitiva GcLeave.....	49
Figura 4.11 – Funcionamento da primitiva GcPos.....	50
Figura 4.12 – Funcionamento da primitiva GcSize.....	50
Figura 4.13 – Funcionamento da primitiva GcSend.....	51
Figura 4.14 – Funcionamento da primitiva GcReceive.....	52
Figura 5.1 – Visão geral do ambiente de comunicação em grupo.....	54
Figura 5.2 – Protótipo do mecanismo de comunicação em grupo.....	55
Figura 5.3 – Servidor de grupos.....	56

ÍNDICE DE TABELAS

Tabela 3.1 – Primitivas de comunicação do ISIS.....	30
Tabela 3.2 – Primitivas de gerenciamento dos grupos no GPS.....	34
Tabela 3.3 – Primitivas de comunicação no GPS.....	34
Tabela 3.4 – Tabela de grupos mantida pelo servidor de grupos.....	35
Tabela 4.1 – Primitivas de gerenciamento de processos.....	39
Tabela 4.2 – Primitivas de comunicação entre processos.....	40
Tabela 4.3 – Primitivas de controle do NC.....	40
Tabela 4.4 – Base de dados do servidor SG.....	43
Tabela 4.5 – Primitivas da camada GNCG.....	45
Tabela 4.6 – Primitivas da camada CG.....	45

LISTA DE ABREVIações

SISD – *Single Instruction, Single Data*

MISM - *Multiple Instruction, Single Data*

SIMD – *Single Instruction, Multiple Data*

MIMD – *Multiple Instruction, Multiple Data*

Mbps – Megabits por segundo

PCs – *Personal computer*

LAN – *Local Area Network*

RPC – *Remote Procedure Call*

CPU – *Central Process Unit*

WAN – *Wide Area Network*

NACK – *Negative ACKnowledgement*

ACK – *ACKnowledgement*

FLIP – *Fast Local Internet Protocol*

NT – Nó de trabalho

NC – Nó de controle

NCE – Nó de comunicação externa

RAM – *Random Access Memory*

GBS – Gerenciamento do barramento de serviços

GP – Gerenciamento de processos

CP – Comunicação de processos

GNC – Gerenciamento de nós e conexões

CG – Comunicação em grupo

GNCG - Gerenciamento de nós, conexões e grupos

SG – Servidor de Grupos

API – *Application Programming Interface*

LMP – Laboratório de Mecânica de Precisão

LacPad – Laboratório de Computação Paralela e Distribuída

MHz – Mega Hertz

MB – Mega bytes

GCC – Gnu C Compiler

SC – Servidor de conexões

RESUMO

O objetivo principal deste trabalho é propor um mecanismo de comunicação entre processos baseado no modelo de comunicação em grupo para o ambiente paralelo/distribuído CRUX.

Um programa paralelo/distribuído no ambiente CRUX é composto por vários processos que se comunicam através de mensagens (comunicação um-para-um). Os processos são criados dinamicamente, podendo ser mapeados em qualquer um dos componentes (nós) do ambiente paralelo/distribuído CRUX. A partir do mecanismo proposto será possível: a) comunicar (envio de mensagens) um processo com n outros processos pertencentes ao mesmo grupo (comunicação um-para- n); b) uma melhor estruturação das aplicações que têm comunicação em grupo como requisito e c) um melhor desempenho na comunicação entre os grupos de processos.

O mecanismo proposto tem como metas principais: suportar o gerenciamento dos grupos e o envio/recepção de mensagens de/para os grupos. Para tanto, utiliza uma abordagem onde um servidor de grupos centralizado mantém uma tabela com informações sobre os grupos do sistema. As mensagens podem ser enviadas ponto-a-ponto para um membro de um grupo ou então difundidas para todos os membros do grupo. Em qualquer um dos casos a comunicação é síncrona.

PALAVRAS-CHAVE: Comunicação em grupo, sistemas operacionais distribuídos e programação paralela.

ABSTRACT

The main objective of this work is to propose an interprocess communication mechanism, based on the group communication model, for the CRUX parallel/distributed environment.

A parallel/distributed program, in CRUX environment, is composed by several processes that communicate through messages (communication one-to-one). Processes are dynamically created, and could be mapped in anyone of the components (node) of CRUX parallel/distributed environment. Starting from the proposed mechanism it will be possible a) the communication (sending/receiving of messages) from a process to n other processes belonging to the same group (communication one-to- n); b) a better structuring of the applications which have group communication as requirement and c) a better performance in the communication among process groups.

The proposed mechanism has as main goals: to support the management of the groups and sending/receiving of messages from/to the groups. In addition, it uses a centralized group server which maintains all the informations about all the groups. The messages can be sent point-to-point to a member of a group or be broadcast to all the members of the group. In both cases the communication is synchronous.

KEYWORDS: Group communication, distributed operating system, parallel programming

SUMÁRIO

1	INTRODUÇÃO.....	1
2	CONCEITUAÇÃO BÁSICA.....	3
	2.1 Organização de Hardware em Sistemas Distribuídos.....	3
	2.1.1 Multiprocessadores.....	4
	2.1.2 Multicomputadores.....	5
	2.1.3 Cluster.....	7
	2.2 Aspectos Básicos de Modelos de Programação Paralela.....	8
	2.2.1 Paralelismo.....	9
	2.2.2 Criação e Término de Processos.....	10
	2.2.3 Mapeamento.....	11
	2.3 Comunicação entre Processos.....	12
	2.3.1 Memória Compartilhada.....	12
	2.3.2 Troca de Mensagens.....	13
	2.3.3 Modelos de Troca de Mensagens.....	15
3	COMUNICAÇÃO EM GRUPO.....	19
	3.1 Atomicidade.....	19
	3.2 Ordenação de Mensagens.....	20
	3.3 Endereçamento.....	21
	3.4 Formas de Comunicação em Grupo.....	21
	3.5 Sobreposição de Grupos.....	24
	3.6 Classificação de Grupos.....	24
	3.7 Gerenciamento de Grupos.....	26
	3.8 Trabalhos Relacionados.....	26
	3.8.1 AMOEBA.....	27
	3.8.2 ISIS.....	29
	3.8.3 TRANSIS.....	31
	3.8.4 GPS (Group based Parallel System)	33
4	MECANISMO DE COMUNICAÇÃO EM GRUPO GcCRUX.....	36
	4.1 Ambiente Paralelo/Distribuído CRUX.....	36

4.2	Comunicação em Grupo no GcCRUX.....	41
4.2.1	Arquitetura do Mecanismo GcCRUX.....	41
4.2.2	Suporte à Comunicação.....	42
4.2.3	Gerenciamento de Grupos.....	42
4.2.4	Ordenação e Entrega de Mensagens.....	43
4.2.5	Primitivas do Mecanismo de Comunicação em Grupo.....	44
5	IMPLEMENTAÇÃO DO PROTÓTIPO.....	54
5.1	Descrição do Protótipo.....	55
5.2	Servidor de Conexões.....	56
5.3	Exemplo de Aplicação.....	59
6	CONCLUSÕES.....	62
6.1	Perspectivas Futuras.....	63
7	REFERÊNCIAS.....	64

1 INTRODUÇÃO

Com o advento de novas tecnologias nos dias atuais, onde computadores tornaram-se acessíveis, houve uma intensa corrida, onde pequenas e médias empresas e até usuários comuns adquiriram seus equipamentos e montaram suas redes de computadores. A informação tornou-se mais acessível e grande parte das pesquisas acadêmicas da atualidade baseiam-se em formas de facilitar, agilizar e organizar as informações em redes de computador.

Existem aplicações que necessitam de desempenho muito grande por parte dos processadores onde rodam. Se estas aplicações forem executadas seqüencialmente tornam-se pouco eficientes, necessitando então, serem processadas paralelamente para melhorar o desempenho. Uma solução é dividir uma tarefa em tarefas menores executando-as paralelamente em diferentes processadores. Este é o principal objetivo do processamento paralelo/distribuído: melhorar o desempenho das aplicações que necessitam de grande poder computacional e que são pouco eficientes quando executadas seqüencialmente.

Para a execução de uma tarefa paralela/distribuída é essencial que haja cooperação entre os processos que compõem o programa paralelo. A forma mais elementar de cooperação entre processos, em ambientes distribuídos do tipo multicomputador, é baseada na troca de mensagens diretas entre dois processos e é chamada comunicação ponto-a-ponto, onde um processo-origem envia uma mensagem para um processo destino. Entretanto, por questões de desempenho e até por facilidade de programação, muitas aplicações paralelas/distribuídas têm como requisito um sistema de troca de mensagens que suporte comunicação em grupo. Considere, por exemplo, um grupo de servidores de arquivos que tem como tarefa comum oferecer um serviço de arquivos tolerante a falhas. Neste caso, espera-se que as requisições de serviço emitidas pelos clientes sejam

recebidas por todos os servidores. Assim, mesmo que um servidor de arquivos não puder atender a requisição, outros servidores poderão atendê-la.

Para que isso seja possível, é necessário que exista um mecanismo de comunicação alternativo, diferente da comunicação envolvendo apenas dois processos, onde seja possível a recepção de uma mesma mensagem por vários recebedores. Este mecanismo e as suas primitivas de controle podem ser fornecidos pelo sistema operacional, através de chamadas de sistema ou biblioteca de funções.

Este trabalho apresenta uma revisão conceitual dos sistemas paralelos/distribuídos, incluindo um estudo de alguns sistemas existentes que suportam comunicação em grupo, e descreve o desenvolvimento de um mecanismo de comunicação em grupo para o ambiente paralelo/distribuído CRUX [CORSO (1993)].

O capítulo 2 apresenta uma visão geral da organização de hardware em sistemas distribuídos. Também descreve alguns conceitos básicos relacionados a programas paralelos e comunicação entre processos. No capítulo 3 é apresentada uma revisão de comunicação em grupo incluindo alguns sistemas existentes que suportam a comunicação em grupo, que foram estudados e usados como exemplos. O capítulo 4 descreve a proposta para um mecanismo de suporte à comunicação em grupo no ambiente CRUX. O capítulo 5 descreve a implementação do protótipo, efetuada no decurso deste trabalho, do mecanismo de comunicação em grupo para o ambiente paralelo/distribuído baseado em multicomputador CRUX, que foi desenvolvida durante este trabalho. Finalmente, o capítulo 6 apresenta as conclusões do trabalho e sugere algumas perspectivas futuras para sua continuação.

2 CONCEITUAÇÃO BÁSICA

2.1 Organização de Hardware em Sistemas Distribuídos

Os sistemas distribuídos caracterizam-se por serem compostos de múltiplos processadores. Assim, é possível classificar os sistemas distribuídos conforme o processamento dos seus dados segundo taxonomia de FLYNN (1972), baseado em dois aspectos: fluxo de dados e fluxo de instrução.

SISD (*Single Instruction, Single Data*) – Um único fluxo de instrução e um fluxo de dados. Todos os computadores com apenas um processador caem nesta categoria.

MISD (*Multiple Instruction, Single Data*) – Vários fluxos de instrução e um fluxo de dados. Na prática, não existe um exemplo concreto deste tipo de sistema distribuído.

SIMD (*Single Instruction, Multiple Data*) – Um único fluxo de instrução, vários fluxos de dados. Alguns supercomputadores se enquadram nesta categoria.

MIMD (*Multiple Instruction, Multiple Data*) – Vários fluxos de instrução e vários fluxos de dados. Multicomputadores, multiprocessadores e as outras arquiteturas paralelas são exemplos práticos deste tipo de sistema distribuído.

TANENBAUM (1995) vai além, sugerindo a divisão dos sistemas MIMD baseado na interação de hardware entre eles, conforme a figura abaixo:

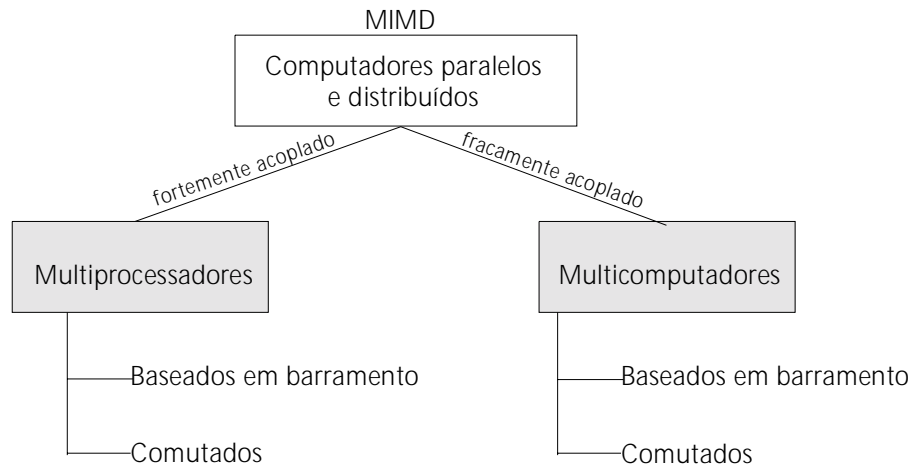


Figura 2.1 - Taxonomia de sistemas paralelos e distribuídos

2.1.1 Multiprocessadores

As máquinas fortemente acopladas são aquelas onde a comunicação é feita através de memória compartilhada, como no caso dos multiprocessadores. Neste caso, o meio de interconexão entre memória e processador pode ser: por barramento ou chaveado.

Baseados em barramento – são compostos de alguns processadores conectados em um barramento comum e que também compartilham um módulo de memória. Neste tipo de configuração, com um número razoavelmente pequeno de processadores o barramento ficará saturado. Para minimizar o *overhead* quando o mínimo de processadores aumenta, utiliza-se memória cache entre o processador e o barramento conforme a Figura 2.2. O uso desta arquitetura permite que o sistema possa ter de 32 a 64 processadores ligados ao barramento [TANENBAUM (1995)].

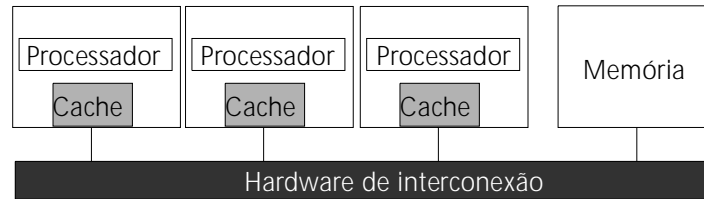


Figura 2.2 – Sistema multiprocessador baseado em barramento

Chaveados – para conectar os processadores de um sistema multiprocessador com mais de 64 processadores, pode-se dividir a memória em módulos, conectando-os aos processadores através de interconexões do tipo *crossbar* ou multi-estágio (por exemplo: rede Ômega) como a Figura 2.3. Cada processador e cada memória têm uma conexão nos sistemas. Com o chaveamento do tipo *crossbar* muitos processadores (depende do número de módulos) podem acessar a memória ao mesmo tempo, porém com n processadores e n memórias, serão necessárias n^2 conexões. O chaveamento do tipo multi-estágio é usado para minimizar este problema.

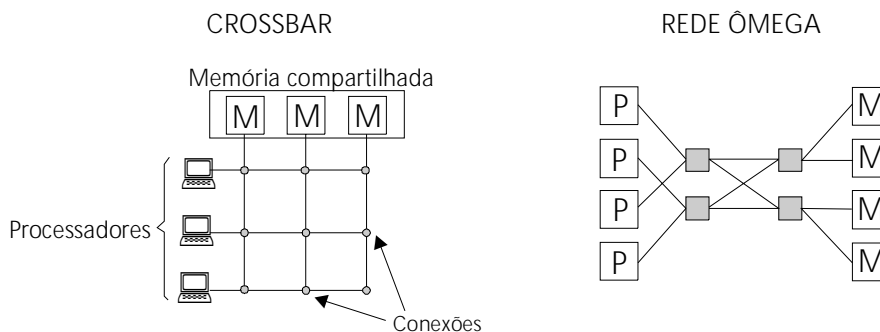


Figura 2.3 – Sistemas multiprocessadores chaveados dos tipos *crossbar* e rede ômega

2.1.2 Multicomputadores

As máquinas fracamente acopladas são caracterizadas pelo fato da comunicação ser feita através de mensagens, sem memória compartilhada, como é o caso dos multicomputadores. Este tipo de sistema fracamente acoplado melhor caracteriza os sistemas distribuídos.

Cada processador tem um acesso direto e exclusivo a sua memória local. Além disso, podem ser divididos de acordo com a interação de hardware. Neste caso, a interconexão que ocorre entre os processadores pode ser baseada em barramento ou chaveada.

Baseados em barramento – a Figura 2.4 mostra um exemplo onde geralmente existe pouco tráfego sobre o barramento, pois cada processador tem sua memória privativa, sendo que não existe a necessidade de implementar um barramento de altíssima velocidade. Este sistema pode ser interconectado através de uma rede local, cuja velocidade é de 10 a 100 Mbps.

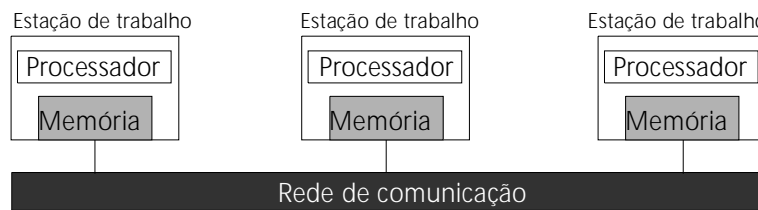


Figura 2.4 – Sistema multicomputador baseado em barramento

Chaveados – a Figura 2.5 mostra as topologias de grelha e hipercubo. As grelhas são fáceis de entender e de implementar em placas de circuito impresso. No hipercubo, cada vértice é um processador e a união entre vértices, uma conexão entre dois processadores.

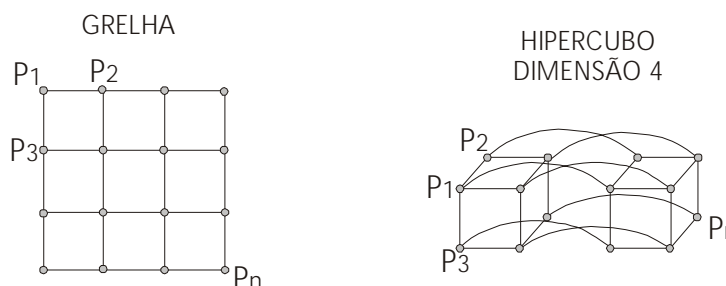


Figura 2.5 – Sistemas multicomputadores comutados dos tipos grelha e hipercubo

Os sistemas multiprocessadores baseados em barramento são limitados pela quantidade máxima de processadores e memórias suportados. As redes *crossbar* são de custo altíssimo e as redes ômega além de dispendiosas, são lentas.

A construção de um sistema multiprocessador fortemente acoplado com uma grande memória compartilhada é uma tarefa difícil e bastante dispendiosa.

2.1.3 Cluster

Um *cluster* é um tipo de sistema de processamento paralelo e distribuído, o qual consiste de uma coleção de computadores independentes interconectados trabalhando juntos como um único recurso integrado [BUY YA (1999)]. Um nó pode ser um único sistema ou um multiprocessador (PCs, estações de trabalho, ou SMPs) com memória, entrada e saída e um sistema operacional. Os *clusters* referem-se geralmente a dois ou mais computadores conectados. Os nós podem existir em um único local ou podem ser fisicamente separados e conectados via LAN, como mostra a Figura 2.6. Um *cluster* de computadores baseado em LAN parece um sistema único para os usuários e suas aplicações [BUY YA (1999)].

As necessidades de comunicação de aplicações distribuídas são diversas e variam de comunicação confiável ponto-a-ponto à comunicação *multicast* não-confiável. A infraestrutura das comunicações necessita de suporte a protocolos que são usados para transporte de dados, fluxo de dados, comunicação em grupo e em objetos distribuídos.

Os serviços de comunicação empregados fornecem os mecanismos básicos necessários por um *cluster* para o transporte administrativo e de dados de usuário. Estes serviços também fornecerão ao *cluster* uma qualidade importante de parâmetros de serviço, como latência, largura de banda, confiabilidade e tolerância à falhas [BUY YA (1999)]. Tipicamente, os serviços de rede são designados como uma pilha hierárquica de protocolos.

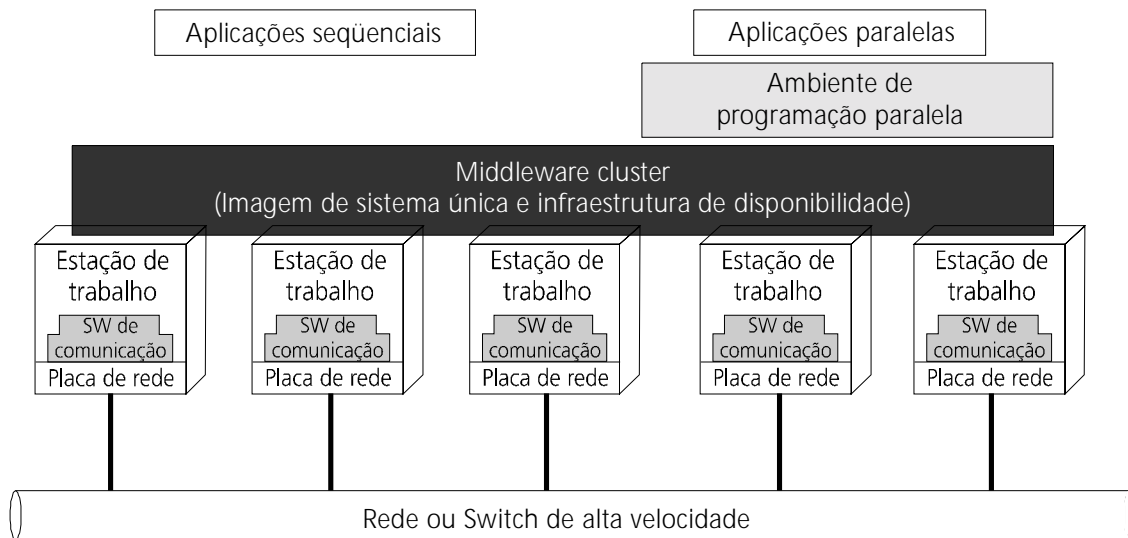


Figura 2.6 – Estrutura da arquitetura *cluster*

Tradicionalmente os serviços de comunicação do sistema operacional (*pipes/sockets*) têm sido usados para comunicação entre processos em sistemas de troca de mensagem. Como resultado, a comunicação entre origem e destino envolve operações dispendiosas, como as trocas de mensagens entre muitas camadas, cópia de dados, checagem de proteção e medidas de comunicação confiável [BUYA (1999)].

2.2 Aspectos Básicos de Modelos de programação Paralela

Um modelo de programação paralela define os mecanismos que poderão ser usados pelo programador para a execução paralela de programas. A execução paralela envolve basicamente: o particionamento de um programa em tarefas menores e o mapeamento das mesmas nos processadores do sistema (expressão do paralelismo); a troca de dados entre as diferentes tarefas do programa (comunicação) e a sincronização na execução das mesmas [TANENBAUM (1992)].

2.2.1 Paralelismo

Um modelo de programação paralela deve oferecer mecanismos para se dividir um programa em tarefas menores, que possam ser executadas simultaneamente, em diferentes processadores. Define-se unidade de paralelismo como sendo a mínima parte de um programa capaz de ser executada em paralelo [TANENBAUM (1992)]. A unidade de paralelismo pode variar desde um processo até mesmo uma simples instrução de programa [BAL (1989)]. Assim, um programa paralelo pode então ser considerado como um conjunto de processos que cooperam entre si para a realização de uma dada tarefa.

Em um programa paralelo, os processos podem ser criados de forma explícita ou implícita [BAL (1989)]. Na criação implícita, utilizada em algumas linguagens de programação, os processos são associados a um tipo de dado e são instanciados quando uma variável deste tipo é declarada no programa. Assim, os usuários podem escrever seus programas em linguagens de programação seqüenciais, mas a análise do programa fonte para encontrar partes do programa que possam ser executadas em paralelo é de responsabilidade do compilador. Com o paralelismo explícito, o programador é responsável pela expressão do paralelismo no seu programa, representando os processos e suas interações. A criação explícita é expressa através de uma função, designada especialmente para realizar esta tarefa. Assim, o programador pode produzir programas paralelos mais eficientes por ter conhecimento da aplicação, embora o uso de construções paralelas acrescente um grau de dificuldade para a produção e correção de programas [MERKLE (1996)].

O processamento paralelo de programas em um multicomputador requer a utilização de técnicas de decomposição [HWANG (1993)] para dividir, mapear e garantir o equilíbrio da carga computacional e estruturas de dados.

Um processo pode ser definido como uma entidade totalmente independente, que executa código de programa e que possui um estado próprio e dados [BAL (1989)]. A execução do código geralmente é realizada de forma seqüencial, no entanto, um processo pode apresentar vários fluxos de execução ativos, também chamados de *threads* [TANENBAUM (1992)]. As *threads* associadas a um mesmo processo executam conjuntos de instruções diferentes, mas compartilham a memória do processo, tendo acesso direto a todos os dados definidos globalmente pelo mesmo. Em um ambiente multiprocessador, as *threads* de um processo podem ser executadas nos diferentes processadores da máquina.

2.2.2 Criação e Término de Processos

A criação de processos pode ser classificada como estática ou dinâmica. Quando a criação é estática, o conjunto de processos que irão compor o programa paralelo deve ser definido antes da execução, e não pode ser alterado até o fim do programa. Uma desvantagem deste tipo de criação é exigir que o número total de processos deva ser conhecido de antemão, o que pode não ser possível, dependendo das características da aplicação que será implementada. Quando a criação é dinâmica, os processos podem ser instanciados a qualquer momento durante a execução do programa [MACHADO (1997)].

O término de processos também é uma questão importante, ligada diretamente à criação de processos. Quando um processo termina de executar, podem ocorrer três tipos de situação: o término do processo causa automaticamente a morte de todos os seus processos filhos; o processo se bloqueia esperando até que os seus filhos terminem ou quando um processo termina, seus filhos se tornam filhos do processo *init*.

2.2.3 Mapeamento

Todos os processos criados em um programa paralelo devem ser associados a algum processador do sistema. Esta associação de processos a processadores é chamada de mapeamento ou *mapping* [BAL (1989)]. O mapeamento deve levar em conta o número de processadores disponíveis e a maneira de distribuir os processos nestes processadores de forma a se obter o maior desempenho possível na execução de uma aplicação. Como exemplo, uma técnica geralmente utilizada quando o número de processos é maior que o número de processadores, consiste em mapear para o mesmo processador aqueles processos que interagem intensamente entre si, minimizando assim, o tempo necessário para a troca de informações e melhorando, portanto, o desempenho do programa.

Quando a criação de processos é estática, o mapeamento é definido no momento da carga do programa na máquina paralela. Porém, quando processos são criados dinamicamente, o mapeamento é feito em tempo de execução.

A associação de processos a processadores pode ser realizada implicitamente pelo sistema que suporta a execução de um dado modelo de programa ou então explicitamente pelo próprio programador. No mapeamento implícito, o sistema geralmente utiliza políticas de balanceamento de carga [BAL (1989)] para efetuar uma distribuição mais adequada dos processos aos processadores. No entanto, como o sistema geralmente não possui conhecimentos específicos sobre o funcionamento das aplicações, a distribuição efetuada pode não ser a ideal e até mesmo afetar o desempenho dos programas [BOKHARI (1991)].

Quando o mapeamento fica a cargo do programador, ele deve então decidir, com base no conhecimento que possui da aplicação, a melhor maneira de distribuir os processos pelos nós do sistema.

2.3 Comunicação entre Processos

A maior diferença entre um sistema distribuído e um sistema monoprocessador é a forma de implementar a comunicação entre os processos [TANENBAUM (1992)].

A cooperação entre os processos que compõem um programa paralelo/distribuído é essencial, pois todos trabalham em conjunto para a realização de uma mesma tarefa. Os processos interagem entre si para trocar dados ou então para sincronizarem as suas ações [BAL (1989), FOSTER (1995), KITAJIMA (1995)]. A comunicação entre processos pode ser realizada basicamente de duas formas: através de uma memória compartilhada ou por meio da troca de mensagens.

2.3.1 Memória Compartilhada

Na comunicação via memória compartilhada, os processos de um programa compartilham uma área de memória comum onde dados podem ser lidos e escritos por qualquer processo (Figura 2.7). Neste caso, existem mecanismos de sincronização, tais como *locks*, semáforos e monitores [ANDREWS (1991)] que são utilizados para controlar o acesso concorrente de vários processos a um mesmo dado na memória.

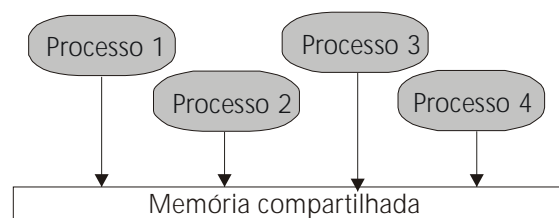


Figura 2.7— Modelo de comunicação via memória compartilhada

Os modelos de programação que suportam o compartilhamento de dados são mais adequados para máquinas paralelas com memória fisicamente compartilhada (o suporte do *hardware* torna mais eficiente a implementação da comunicação em *software*).

2.3.2 Troca de Mensagens

Quando os processos possuem apenas sua própria memória local (não compartilham memória) a troca de dados entre os mesmos é realizada através do envio e recebimento de mensagens: o processo que precisa acessar um dado produzido por outro, deve receber este dado através de uma mensagem enviada pelo processo produtor. A implementação de modelos de programação baseados na troca de mensagens é bastante natural em máquinas paralelas com memória distribuída, como os multicomputadores, onde a comunicação entre os processadores do sistema também é realizada através da troca de mensagens.

A troca de mensagens envolve questões que devem ser analisadas ao se definir um modelo de programação baseado neste tipo de comunicação. Uma das mais importantes é o tipo de nomenclatura usada para a troca de mensagens.

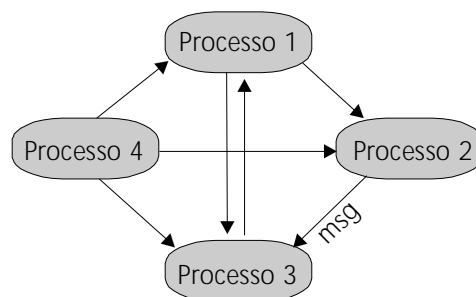


Figura 2.8— Modelo de comunicação via troca de mensagens

Nomeação Direta e Indireta

Na nomeação direta cada processo possui um identificador associado a ele e as mensagens são destinadas a processos específicos. Se a nomeação é indireta, as mensagens são endereçadas a um objeto intermediário, geralmente chamado de caixa postal (*mailbox*) ou porta. Uma caixa postal pode ser associada a um processo específico ou então representar um objeto totalmente autônomo, independente de qualquer processo [MACHADO (1997)].

Não-Characterização

A não-caracterização é a propriedade que permite a um processo interagir com um conjunto de outros processos, sem precisar identificar especificamente cada um deles. Esta propriedade torna a troca de mensagens mais flexível. Em alguns casos, no entanto, além de se expressar a não-caracterização, é necessário também controlar de alguma forma o conjunto de processos com o qual se deseja interagir. Como exemplo, o processo que gerencia um depósito de dados limitado, somente pode receber dados de um conjunto de processos produtores para serem armazenados no depósito quando este último não estiver cheio, e somente pode atender o pedido de processos consumidores de dados quando o depósito não estiver vazio. Uma notação bastante utilizada para a expressão e controle da não-caracterização é o comando guardado [DIJKSTRA (1975)], implementado geralmente em linguagens de programação distribuída e paralela.

Sincronização

A sincronização estabelece restrições na ordem das operações dos processos interativos [MERKLE (1996)]. Existem dois tipos para se detectar quando uma sincronização é necessária, quando existem duas operações pertencentes a processos:

- 1 - A exclusão mútua, se as operações não precisam ser executadas simultaneamente, utiliza-se a sincronização para implementar o acesso exclusivo ao recurso compartilhado;
- 2 - E a sincronização condicional, se a operação deve ser executada após a outra, então se utiliza a sincronização para que os processos cooperantes entre si executem etapas seqüenciais do mesmo serviço.

A sincronização pode ser obtida através da própria troca de mensagens, pois o processo que precisa de um dado produzido por outro deve esperar pelo recebimento da mensagem contendo o dado desejado. Outro mecanismo bastante usado para se estabelecer a sincronização entre processos é a barreira [MCBRYAN (1994), FOSTER (1995)]. Uma barreira consiste em um ponto de sincronismo que envolve um conjunto de processos. Esta função bloqueia o processo que a executa até que todos os demais processos do conjunto também a tenham executado. Quando o último processo do conjunto chama esta função, os demais são desbloqueados, podendo então prosseguir sua execução normalmente (a operação barreira é particularmente útil em programas paralelos compostos de diferentes etapas de cálculo executados de maneira síncrona).

2.3.3 Modelos de Troca de Mensagens

Os modelos de programação paralela devem definir qual o padrão de troca de mensagens que será utilizado e a maneira de agrupar as características da comunicação. Os principais modelos de troca de mensagens existentes são o cliente/servidor, chamada remota de procedimento (*Remote Procedure Call* -RPC) e a comunicação em grupo. Com exceção do modelo de comunicação em grupo, que suporta a cooperação entre um conjunto de processos, os demais modelos envolvem apenas um processo emissor e um processo receptor da mensagem (comunicação ponto-a-ponto).

Em qualquer um dos modelos acima, a troca de mensagens pode ser síncrona ou assíncrona [TANENBAUM (1992)].

✧ **Comunicação síncrona** - Em uma comunicação síncrona, o canal de comunicação serve ao transporte de mensagens entre os dois processos envolvidos. Os canais não possuem capacidade de armazenamento e as comunicações requerem a participação simultânea dos processos envolvidos. O processo-emissor é bloqueado até que o receptor aceite a mensagem, e o processo receptor espera até que a mensagem desejada chegue. Desta forma, os processos não só trocam dados como também sincronizam suas execuções.

✧ **Comunicação assíncrona** - Em uma comunicação assíncrona, o canal serve ao transporte bidirecional de mensagens entre os processos envolvidos. Os canais possuem capacidade de armazenamento e as comunicações não requerem a participação simultânea dos processos envolvidos. O processo-emissor não espera até que o processo-receptor receba a mensagem: depois de enviá-la, o emissor prossegue imediatamente a sua execução. O processo receptor não bloqueia se a mensagem desejada não se encontra disponível. Neste caso, o processo retoma sua execução normal e de tempos em tempos pode chamar a função de recepção novamente até que a mensagem desejada tenha chegado. As mensagens enviadas por um processo a outro geralmente são entregues ao destinatário conforme a ordem de envio. No entanto, esta ordem pode ser alterada caso o processo receptor dê prioridade para receber mensagens com um determinado tipo, por exemplo.

Modelo Cliente/Servidor

O modelo cliente/servidor é baseado num protocolo de solicitação/resposta para evitar o tráfego considerável dos protocolos orientados à conexão. É um modelo simples, eficiente, não precisa de conexão e fácil de implementar. Devido a esta estrutura, os serviços de comunicação fornecidos pelo *kernel* podem, por exemplo, ser reduzidos a duas chamadas de sistema, uma para envio outra para recepção de mensagens. A idéia é estruturar o sistema operacional como um

grupo de processos cooperantes chamados servidores que oferecem serviços a processos chamados clientes [TANENBAUM (1992)].

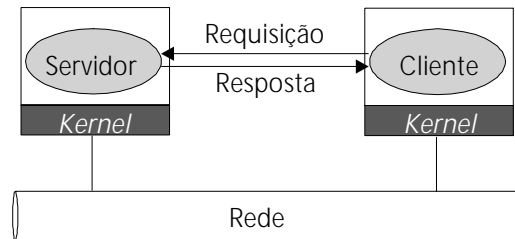


Figura 2.9 - Modelo cliente/servidor (envio/resposta)

Usualmente, o cliente e o servidor rodam no mesmo *kernel* com processos rodando como processos usuários.

Chamada Remota de Procedimento - RPC

Uma meta almejada por todo sistema distribuído é a de se parecer muito com um sistema centralizado (para os usuários). Apesar do modelo cliente/servidor ser conveniente, operações de entrada e saída não são conceitos-chaves dos sistemas centralizados, por isso utilizá-los pode levar a cometer erros graves [TANENBAUM (1995)]. BIRRELL e NELSON (1984) permitiram que programas chamassem procedimentos localizados em outras máquinas. Este método é conhecido como chamada remota de procedimento, RPC.

Na Figura 2.10, que esquematiza um modelo RPC, os *stubs* têm a função de empacotar e desempacotar parâmetros e resultados.



Figura 2.10 – Chamada remota de procedimento RPC [TANENBAUM (1992)]

Existem alguns problemas relacionados com a este esquema. O processo-chamador e os procedimentos chamados usualmente rodam em máquinas diferentes, conseqüentemente rodam em espaços de endereçamento diferentes. Por isso, a passagem de quaisquer parâmetros e dos resultados pode ser complicada (principalmente a passagem de ponteiros).

A RPC consiste em um mecanismo de comunicação de mais alto nível, similar a uma chamada normal de função. Quando um processo realiza uma chamada remota de procedimento, os parâmetros de entrada para a função são enviados através de uma mensagem de requisição para outro processo (possivelmente residente em outro processador) que será então o responsável por executá-la. Posteriormente, os resultados obtidos com a execução da função são retornados, através de uma mensagem de resposta, para o processo-solicitante. Enquanto a função está sendo executada pelo processo-destino, o solicitante permanece bloqueado, na espera da mensagem de resposta. A chamada remota de procedimento é o modelo de comunicação característico de aplicações do tipo cliente-servidor [COULOURIS, DOLLIMORE e KINDBERG (1994)], nas quais processos clientes dependem de serviços realizados por processos servidores.

A comunicação em grupo de processos, pela importância neste trabalho, será introduzida no próximo capítulo.

3 COMUNICAÇÃO EM GRUPO

Um grupo é um conjunto de processos cooperantes (que agem juntos) especificados pelo sistema ou por um usuário [TANENBAUM (1995)].

O principal resultado do conceito de grupo é permitir que processos tratem conjuntos de processos como uma simples abstração. Assim, um processo pode enviar uma mensagem para um grupo sem precisar saber quantos processos compõem o grupo, ou onde eles, os processos, estão executando (pode mudar de uma chamada para outra).

3.1 Atomicidade

Na comunicação em grupo duas propriedades são importantes. A primeira delas é a propriedade de atomicidade (*broadcast* atômico) conhecida como a propriedade do tudo ou nada, onde as mensagens enviadas para um grupo devem ser recebidas corretamente por todos os membros do grupo ou não serem recebidas por nenhum deles. Entretanto, esta propriedade não é facilmente implementada devido ao fato que pode haver quebra na comunicação. Além disso, a garantia desta propriedade tem um custo significativo para o sistema. Por exemplo, um processo-emissor inicia enviando uma mensagem para todos os membros do grupo. Temporizadores são usados e pode haver retransmissões de mensagens quando necessárias. Quando um processo recebe uma mensagem, se ele ainda não tiver recebido esta mensagem em particular, ele também envia para todos os membros do grupo (novamente com temporizadores e retransmissões, se necessário). Se ele já tiver recebido a mensagem, então a mensagem é descartada. Assim, não importa quantas máquinas falharem ou quantos pacotes serão perdidos, todos os processos receberão a mensagem [TANENBAUM (1995)].

3.2 Ordenação de Mensagens

A segunda propriedade é a ordenação de mensagens. O ideal é que implementação garanta que as mensagens cheguem na mesma ordem em que foram enviadas.

A ordenação de mensagens, ilustrada na Figura 3.1, pode ser classificada de três maneiras:

- ordenação por tempo global (ou sistema síncrono) onde as mensagens são repassadas imediatamente e na exata ordem em que elas são enviadas não considerando o tempo envolvido no envio da mensagem. Este tipo de ordenação não é possível de se implementar.
- ordenação por tempo total/consistente (ou sistema fracamente síncrono): entrega as mensagens na mesma ordem que elas chegam, garantidamente todos os processos recebem na mesma ordem. Na ordenação por tempo consistente, as mensagens são enviadas ao mesmo tempo e, de alguma forma, a seqüência é determinada por um árbitro, que considera uma como a primeira e envia para o destino
- ordenação causal (ou sistema virtualmente síncrono) onde as mensagens são relacionadas por uma causalidade, ou seja, um determinado evento pode influenciar o comportamento de um segundo evento, dependendo da ordem em que acontece.

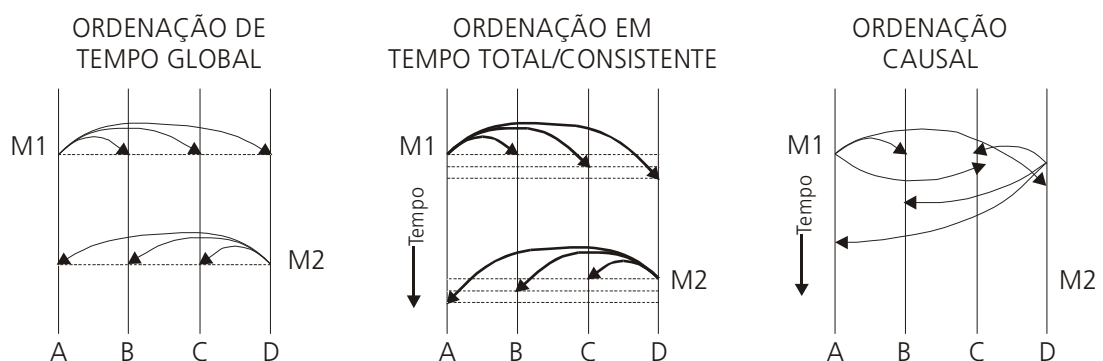


Figura 3.1 - Tipos de ordenação de mensagens

3.3 Endereçamento

Para que as mensagens sejam recebidas pelos grupos, eles precisam ser endereçados. A maneira como o endereçamento é implementado é dependente de *hardware* e pode ser através de *multicast*, *broadcast* ou *unicast*. Se o *hardware* suportar *multicast*, um grupo é tratado como um endereço *multicast* sendo, portanto, enviada a todas as máquinas que fazem parte deste endereço.

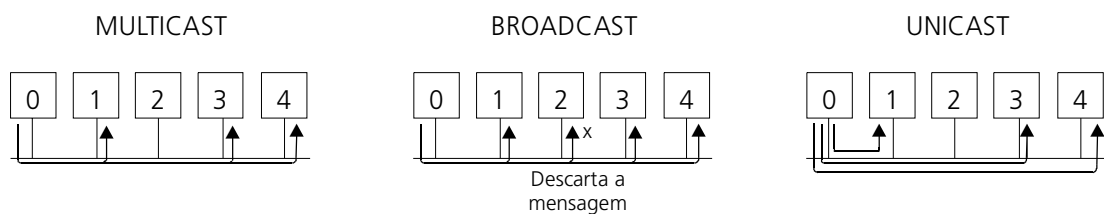


Figura 3.2 - Tipos de endereçamento

Se o *hardware* somente suportar *broadcast*, a mensagem pode ser enviada por *broadcast* para todas as máquinas. Se nenhum dos processos desta máquina for membro do grupo, a mensagem será descartada, senão será entregue ao grupo. Cada vez que houver o recebimento de um pacote, o *software* tem que verificar se o pacote recebido é destinado a ele ou não (esta verificação gasta tempo, tornando esta técnica menos eficiente).

Caso nenhuma destas técnicas esteja disponível, a comunicação pode ser implementada fazendo com que o transmissor envie um pacote para cada um dos membros do grupo, também chamado de *unicast*, como mostrado na Figura 3.2.

3.4 Formas de Comunicação em Grupo

Várias formas de comunicação em grupo são possíveis, podendo-se destacar a comunicação um-para-todos, todos-para-um e a todos-para-todos

Na comunicação um-para-todos também chamada de difusão [TANENBAUM (1995)], um dos processos membro do grupo envia dados para o restante do grupo. Já a comunicação todos-para-um ocorre no sentido contrário, ou seja, os processos pertencentes a um mesmo grupo enviam uma mensagem (resposta) para um único processo destinatário. Na comunicação todos-para-todos cada membro de um grupo envia uma mensagem para os demais componentes do grupo. Da mesma forma que na troca de mensagens ponto-a-ponto, a difusão de mensagens para um grupo e a recepção de mensagens produzidas por membros de um grupo pode ser síncrona ou assíncrona.

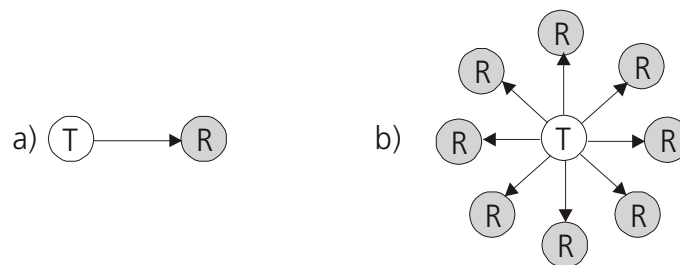


Figura 3.3 – a) Comunicação ponto-a-ponto e b) comunicação um-para-muitos

As mensagens enviadas por um processo a outro geralmente são entregues ao destinatário conforme a ordem de envio. No entanto, esta ordem pode ser alterada caso o processo-receptor dê prioridade por receber mensagens com um determinado tipo, por exemplo.

Entre as principais vantagens do modelo de comunicação em grupo pode-se citar a maior facilidade de expressão da comunicação envolvendo múltiplos processos, assim como, o melhor desempenho que este tipo de comunicação pode oferecer em comparação à troca de mensagens ponto-a-ponto. A primeira vantagem pode ser claramente evidenciada através de um exemplo: admitindo-se que, em um programa paralelo, uma mensagem deva ser enviada para dez processos e que apenas primitivas ponto-a-ponto possam ser utilizadas; o processo emissor, neste caso, terá que enviar uma mensagem especificamente para cada processo-destino. Assim, o emissor terá que efetuar dez vezes o envio

de uma mensagem ponto-a-ponto (Figura 3.4). No entanto, se os processos destino forem incluídos em um grupo, basta para o emissor enviar uma única mensagem (de difusão) para o grupo e todos os processos a receberão.

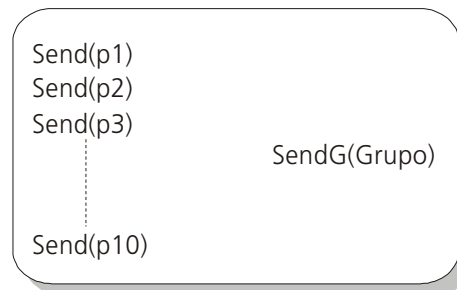


Figura 3.4 – Comunicação ponto-a-ponto versus comunicação em grupo [MACHADO (1997)].

Existe uma desvantagem neste processo, o emissor não pode modificar o *buffer* da mensagem até que a mensagem tenha sido enviada. Existem conseqüências no caso de processos que sobrescrevem a mensagem durante a transmissão. O processo emitido desconhece o fato de que a transmissão foi feita, então ele nunca sabe quando é seguro usar o *buffer*.

Para solucionar estes problemas, podemos usar duas técnicas:

1 – Manter no *kernel* uma cópia da mensagem em um *buffer* interno que permita que o processo continue. Do ponto de vista do emissor, é semelhante à chamada síncrona, assim que o controle volte, ele pode reusar o *buffer*. A desvantagem deste método é que toda mensagem é copiada para o *buffer* do *kernel*. Com muitas interfaces de rede, a mensagem terá que ser copiada para o *buffer* de transmissão de *hardware*.

2 – Interromper o emissor quando uma mensagem tiver sido enviada para informar que o *buffer* está disponível. Não tem cópia no *kernel*, mas as interrupções do nível de usuário fazem a programação ficar difícil e sujeitas a condições de corrida. Este método é considerado mais produtivo e permite uma melhor exploração do paralelismo, mas as desvantagens são maiores do que as vantagens.

3.5 Sobreposição de Grupos

Um processo pode ser membro de múltiplos grupos ao mesmo tempo, podendo causar uma certa inconsistência. Suponha que os processos A e D queiram enviar mensagens para seu grupo usando *unicast*. O processo B receberá uma mensagem do processo A seguida de uma mensagem do processo D e o processo C receberá na ordem inversa, como mostra a Figura 3.5.

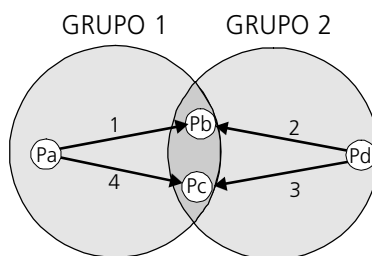


Figura 3.5 - Sobreposição de grupos

3.6 Classificação dos Grupos

Os grupos podem ser organizados de várias maneiras. Grupos abertos ou fechados, pares ou hierárquicos e estáticos ou dinâmicos.

Quanto a sua forma de comunicação, ou seja, quem está enviando para quem, os grupos podem ser fechados, onde apenas os membros do grupo podem enviar para o grupo; ou abertos, onde qualquer processo do sistema pode enviar para qualquer grupo, como mostra a Figura 3.6.

Os grupos fechados são tipicamente usados em processamento paralelo. O grupo interage para resolver o problema, não tem contato com o mundo exterior. Mas quando a idéia de grupos é usada para suportar servidores replicados, os processos não-membros devem poder enviar para o grupo e os membros devem poder

enviar para processos não-membros. Um exemplo de aplicação que utiliza grupos fechados é um jogo de xadrez.

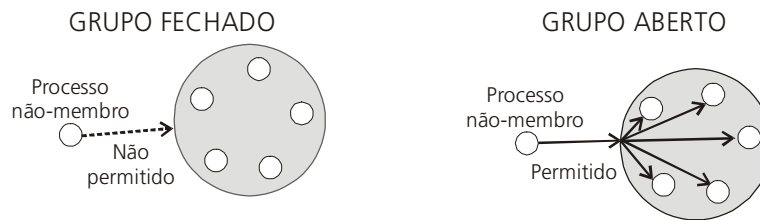


Figura 3.6 - Grupos abertos ou fechados.

Quanto à estrutura interna dos grupos, ilustrado na Figura 3.7, em alguns grupos todos os processos são iguais e as decisões são tomadas coletivamente, a estes grupos chamamos de grupos pares. É um grupo simétrico e não tem ponto de falha. Se um processo falha, o grupo simplesmente fica menor, mas pode continuar. A desvantagem é que a tomada de decisão é mais complicada, pois tem que ser votada, gerando tempo e *overhead*.

Os grupos hierárquicos têm um coordenador que toma as decisões e os outros processos apenas acatam. A desvantagem é que se o coordenador falha, o grupo se torna incapaz de tomar decisões.



Figura 3.7 - Grupos pares ou hierárquicos

Os grupos também podem ser classificados em estáticos ou dinâmicos. Em um grupo estático, o tamanho do grupo é conhecido na sua criação. O grupo termina quando todos os processos terminarem de executar, pois os processos não deixam ou se unem ao grupo. Em grupos dinâmicos não existe um tamanho máximo, os processos são adicionados e excluídos do grupo em qualquer parte da

execução. Para um processo fazer parte do grupo, basta se apresentar como membro.

3.7 Gerenciamento dos Grupos

Para tornar viável a comunicação em grupo, alguma forma de gerenciamento dos grupos (criação/remoção dos grupos e entrada/saída de processos em grupos) é necessária.

Existem duas abordagens principais para este assunto: servidor de grupo e gerenciamento distribuído. Na abordagem servidor de grupo, todas as requisições de tratamento de grupos são enviadas para o servidor que mantém uma base de dados completa de todas as informações necessárias de todos os grupos e seus membros.

Se o servidor falhar, deixa de existir o gerenciamento de processos. Quando o servidor se recuperar, os grupos serão re-iniciados, interrompendo o trabalho que estava em andamento. Este é um método simples, eficiente e fácil de implementar. Na abordagem distribuída, não existe um servidor de grupos, sendo que todas as requisições relativas aos grupos devem ser submetidas a todos os membros do grupo. Embora pareça simples, a implementação desta abordagem é complexa, começando pela necessidade de todos os membros do grupo manterem informações sobre os grupos.

3.8 Trabalhos Relacionados

Nesta seção são apresentados alguns sistemas que suportam a comunicação em grupo. Os sistemas que estão sendo levados em consideração implementam o conceito de grupo fornecendo serviços, desde o gerenciamento dos grupos até o envio/recepção de mensagens, incluindo protocolos de

ordenação e entrega. Os sistemas revisados incluem: AMOEBA, ISIS, TRANSIS e GPS.

3.8.1 AMOEBA

O Amoeba [AMOEBA, KAASHOEK (1993), TANENBAUM e (1991)] é um sistema operacional distribuído desenvolvido em Amsterdã, Holanda, na Universidade de Vrije. É um sistema baseado em *pool* de processadores onde as máquinas no *pool* podem ser de diferentes arquiteturas (Figura 3.8).

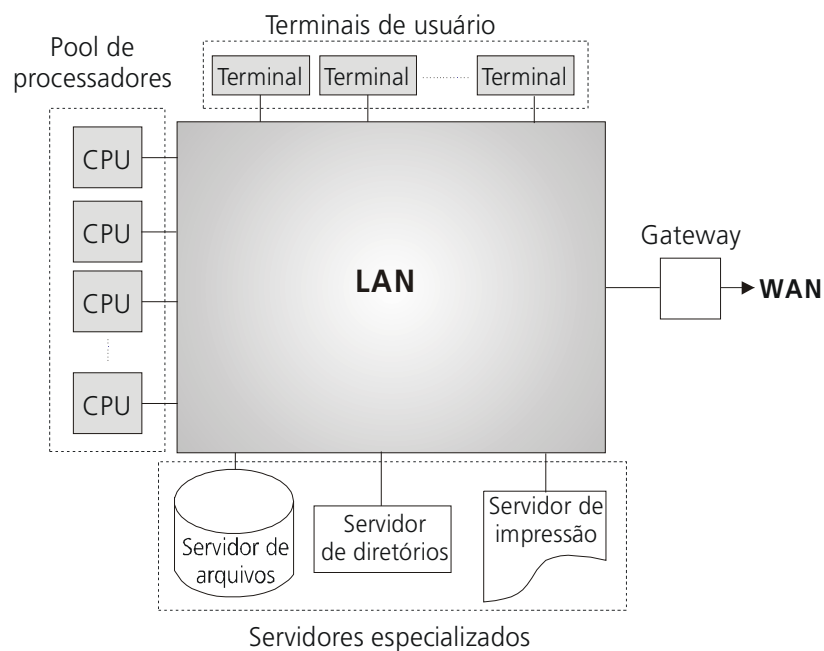


Figura 3.8 – Arquitetura de *hardware* do AMOEBA

O Amoeba utiliza um protocolo com ordenação total consistente [ANDREWS (1991)], cujo responsável pela ordem das mensagens é o seqüenciador. O processo origem envia a mensagem para o seqüenciador, que insere um número de seqüência e envia a mensagem para todos os membros do grupo, usando uma mensagem *broadcast*. Os números da seqüência são utilizados para assegurar que as mensagens sejam entregues na mesma ordem para todos os membros.

Quando um processo-destino (membro do grupo) perde uma mensagem do seqüenciador, ele requisita a retransmissão enviando uma mensagem de NACK (ACK negativo).

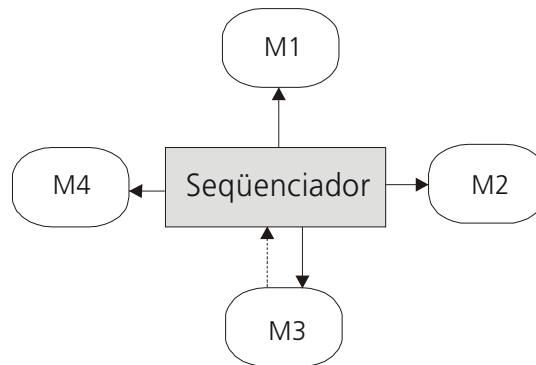


Figura 3.9 – Protocolo com seqüenciador do AMOEBA

O Amoeba suporta duas formas de comunicação: RPC, usando troca de mensagens *unicast* (ponto-a-ponto) e comunicação em grupo. A comunicação em grupo utiliza o *broadcast* por *hardware* ou *multicast* se este estiver disponível no sistema, senão o *kernel* o simula transparentemente com mensagens individuais.

O Amoeba utiliza o conceito de grupos fechados (apenas membros do grupo podem enviar mensagens para o grupo). A informação a respeito do tamanho do grupo (processos-membros) não é conhecida por um processador fora do grupo. Os processos podem se juntar ou sair de um grupo dinamicamente e podem pertencer a vários grupos ao mesmo tempo.

As primitivas utilizadas pelos grupos são: `CreateGroup`, `JoinGroup`, `LeaveGroup`, `SendToGroup`, `ReceiveGroup` e `ResetGroup`.

Quando uma aplicação inicia, uma das máquinas é eleita como seqüenciador, como ilustra a Figura 3.10. As primitivas estão disponíveis no nível de sistema (chamadas de sistema), neste caso pode-se dizer que a abordagem adotada no gerenciamento dos grupos é centralizada.

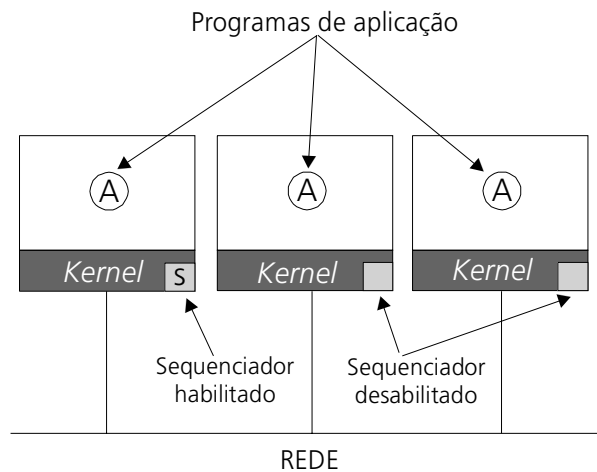


Figura 3.10 - Estrutura do AMOEBA para suporte à comunicação em grupo

O Amoeba funciona melhor em LAN com *broadcast* ou *multicast*. Ele implementa a comunicação em grupo com base em *broadcast* confiável, ou seja, uma mensagem é enviada através da primitiva *SendToGroup*, e será corretamente entregue para todos os membros do grupo, mesmo existindo perda de alguns pacotes.

A comunicação do Amoeba utiliza um protocolo chamado FLIP (*Fast Local Internet Protocol*) para transmissão das mensagens. Ele é utilizado para conseguir um aumento no desempenho com relação ao protocolo UDP/TCP.

3.8.2 ISIS

O ISIS [BIRMAN, JOSEPH e SCHMUCK (1987)] foi desenvolvido na Universidade de Cornell e é uma ferramenta para suporte a programação/construção de aplicações distribuídas e fornece um conjunto de primitivas para a comunicação em grupo (Tabela 3.1) semelhante ao Amoeba. A idéia básica da comunicação em grupo no ISIS é o sincronismo e as primitivas de comunicação são diferentes formas de *broadcast* atômico.

PRIMITIVA	DESCRIÇÃO
ABCAST	fornece comunicação fracamente síncrona e transmite informações entre os membros do grupo;
CBCAST	fornece comunicação; sincronismo virtual e é usada no envio de informações;
GBCAST	usada para gerenciar a filiação de processos a grupos.

Tabela 3.1 – Primitivas de comunicação do ISIS

ABCAST – A primitiva ABCAST utiliza um protocolo de entrega de duas fases com aceitação. Uma mensagem de um processo A recebe um número de seqüência e é enviada para todos os membros do grupo. Cada membro do grupo recebe a mensagem, coloca seu número de seqüência e manda a mensagem de volta. O emissor A então escolhe o maior número de seqüência retornado e envia uma mensagem de aceitação para todos os membros com um número escolhido, o que determina a ordem desta mensagem. Este protocolo garante que todas as mensagens sejam enviadas para todos os processos na mesma ordem.

CBCAST – Utiliza um protocolo onde a garantia de entrega ordenada é implementada apenas para mensagens causalmente relacionadas. Este protocolo funciona da seguinte maneira: em um grupo de n membros, cada processo mantém um vetor com n componentes, um para cada membro do grupo. O i -ésimo componente deste vetor é o número da última mensagem recebida em seqüência do processo i . Os vetores são gerenciados pelo ambiente de execução, e não pelos processos usuários. Quando um processo quer enviar uma mensagem, ele incrementa seu vetor e envia como parte da mensagem.

As condições para a mensagem ser aceita são:

- a) $V_j = L_j + 1$
- b) $V_i \leq L_i \forall i \neq j$

Onde V_i é o i -ésimo componente do vetor na mensagem que está chegando; L_i o i -ésimo componente do vetor na memória do processo receptor e a mensagem foi enviada pelo processo j .

A Figura 3.11 mostra um exemplo de utilização do protocolo CBCAST.

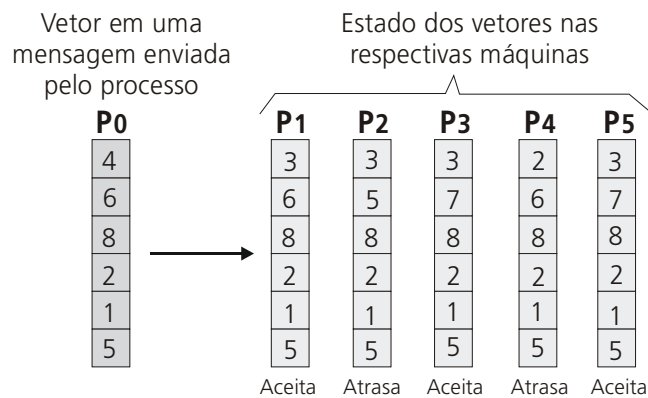


Figura 3.11 – Exemplo do vetor utilizado por CBCAST

GBCAST – Esta primitiva é semelhante a primitiva ABCAST mas em vez de gerenciar o envio de informações, ela é usada para gerenciar a filiação de processos em grupos.

3.8.3 TRANSIS

O Transis [AMIR, DOLEV e KRAMER (1992), DOLEV e MALKI (1995)] foi desenvolvido na *Hebrew University* de Jerusalém. É um pacote da camada de transporte que suporta uma variedade de serviços de troca de mensagens confiáveis (inclusive comunicação em grupo) em topologia de redes. Os grupos são identificados por um nome selecionado pelo usuário e as mensagens são endereçadas para o grupo através deste nome. Os grupos são as unidades básicas da comunicação, onde um processo pode enviar uma mensagem para o grupo e é garantido que todos os membros do grupo recebam.

Os grupos são dinâmicos, os membros deixam o grupo de forma voluntária ou podem ser forçados no caso de falha ou desconexão. Os grupos são abertos, ou seja, qualquer processo pode enviar mensagens para o grupo, independente de

ser membro ou não. O Transis adota esta abordagem com a justificativa de que a comunicação externa a um grupo de processos é muito comum.

Os grupos são criados automaticamente quando o primeiro membro se junta ao grupo, não existe uma primitiva para criação de grupos. Sendo assim, os processos apenas se juntam aos grupos. O mecanismo de gerenciamento de grupos garante que as mudanças nos grupos são informadas e mantidas de forma consistente para todos os membros do grupo. O Transis suporta os conceitos de um modelo de programação baseado em sincronismo virtual [TANENBAUM (1995)], ou seja, garante que a ordem das mudanças nos grupos é consistente em relação à entrega de mensagens regulares.

Com relação aos tipos de entrega/ordenação de mensagens fornecidos pelo Transis:

Não-ordenado: garante entrega de mensagens em todos os destinos ativos, cada processador que recebe a mensagem e entrega a mesma imediatamente para o nível superior.

Ordenação causal: cuja entrega é definida por:

- (1) $m \rightarrow m'$ se $\text{ReceiveG}(m) \rightarrow \text{SendG}(m')$;
- (2) $m \rightarrow m'$ se $\text{SendG}(m) \rightarrow \text{SendG}(m')$.

Ordenação Total: entrega inclui todo tipo de mensagem, seja com relação causal ou concorrente.

Entrega Segura: fornece a informação de que a mensagem foi recebida por todos os destinos (propriedade da atomicidade - tudo ou nada).

Esta última é a mais demorada e dispendiosa porque garante que a entrega é feita após todas as máquinas receberem uma cópia, o que significa que o

emissor espera por ACK de todos os participantes e exige retransmissões no caso de perdas de mensagens.

3.8.4 GPS (Group based Parallel System)

O GPS [MACHADO (1997)] fornece mecanismos para o gerenciamento de grupos, gerenciamento de processos, comunicação e sincronização entre os processos dos grupos.

Suporta grupos estáticos e dinâmicos, abertos e pares. A escolha do tipo de grupo é feita na criação do grupo tendo por base o tipo de aplicação a ser desenvolvida. Os grupos estáticos são gerenciados de forma simples e garantem maior desempenho para as aplicações onde possam ser usados. Os grupos dinâmicos possuem maior tráfego no seu gerenciamento, uma vez que a entrada de processos pode ocorrer a qualquer instante, porém são extremamente úteis para aplicações onde o número de processos varia conforme a carga computacional [MPI-2 (1996)].

No GPS, os grupos são: abertos tornando possível a comunicação entre processos ou membros de grupos diferentes;

são pares, ou seja, não tem hierarquia exceto o processo que tem um filho e ainda os grupos podem ser destruídos explicitamente ou implicitamente, no momento que todos os membros se retirarem.

Os grupos no GPS são gerenciados através das seguintes primitivas:

PRIMITIVAS	PARÂMETROS	DESCRIÇÃO
int ggroupcreate	int grp_id, int nmembers	cria o grupo e o identificador <i>grp_id</i> será associado ao novo grupo enquanto <i>nmembers</i> indica o número de membros que o grupo terá. Se este número for maior do que zero, o grupo será estático, caso contrário será dinâmico
int gpartition	int old_grp_id, int key, int given_rank	uma forma de criar novos grupos. Dependendo da chave cada processo faz parte de um sub-grupo
void ggroup_destroy	int grp_id	O grupo <i>grp_id</i> pode ser destruído de forma explícita com esta função ou de forma implícita, quando detectado que não possuem mais membros ativos
in gjoin	int grp_id, int desired_rank	para que os processos possam fazer parte de um grupo
int gleave	int grp_id	para que os processos deixem o grupo (grupos dinâmicos)
int grank	int grp_id	identifica a posição de um membro dentro do grupo
int gsize	int grp_id	identifica o tamanho do grupo
void gwait	int grp_id, int members	espera até que o grupo tenha um determinado número de membros

Tabela 3.2 – Primitivas de gerenciamento dos grupos no GPS

Os processos interagem entre si através de primitivas *Send* e *Receive*. O envio de mensagens é assíncrono, mas a recepção de mensagens é síncrona, pois o receptor é bloqueado até que a mensagem esteja disponível. O GPS fornece um conjunto de primitivas para o gerenciamento de grupos e para comunicação.

A troca de mensagem é expressa através do uso das seguintes primitivas:

PRIMITIVAS	PARÂMETROS	DESCRIÇÃO
int gsend	int local_grp, int remote_grp, int remote_rank, char* msg, int msglen, int msgtype	Envio assíncrono de mensagens
int grecv	int local_grp, int remote_grp, int remote_rank, char* msg, int msglen, int msgtype, status_t* status	Recepção síncrona de mensagens.
int probe	int local_grp, int remote_grp, int remote_rank, int msgtype, status_t* status	Recepção síncrona e verifica se a mensagem já pode ser recebida.


Tabela 3.3 – Primitivas de comunicação no GPS

A organização dos grupos em GPS utiliza uma abordagem do tipo centralizada. Existe um servidor centralizado associado a um grupo que executa em um nó que é definido por uma função *hash*. Os nomes dos processos de um

grupo são resolvidos a partir de consultas na *cache* local de grupos, quando isto não é possível, o servidor de grupo é acessado. As *caches* são mantidas por um processo gerenciador de *caches*. O servidor de grupo de um nodo mantém uma tabela de grupos contendo informações sobre cada um dos grupos gerenciados pelo servidor. Cada entrada da tabela se refere a um grupo e armazena informações como as da Tabela 3.4.

NOME	DESCRIÇÃO
grp_id	identificador do grupo
grp_struct	identificador do tipo de grupo: estático ou dinâmico
member_list	lista dos identificadores dos processos membros do grupo e seus <i>ranks</i>
grp_size	número de membros do grupo
cur_barrier	estrutura com informações da barreira associada ao grupo
barrier_size_list	lista dos próximos valores a serem associados à barreira do grupo
cache_node_list	lista de informações sobre as caches dos grupos
gwait_list	lista que armazena as requisições <i>gwait</i> pendentes
partition_list	lista sobre os novos grupos criados pelo particionamento

Tabela 3.4 – Tabela de grupos mantida pelo servidor de grupos

Este capítulo abordou uma revisão dos conceitos de comunicação em grupo bem como a descrição de alguns sistemas existentes que suportam comunicação em grupo. Estes sistemas foram usados como base para a descrição do mecanismo e o desenvolvimento do protótipo do  .

4 MECANISMO DE COMUNICAÇÃO EM GRUPO



Originalmente o ambiente CRUX não fornece facilidades para a comunicação em grupo. A meta principal deste trabalho é desenvolver um mecanismo de suporte a comunicação em grupo, pois acreditamos que este mecanismo possibilite um melhor desempenho para as aplicações paralelas que executam no CRUX ou que tenham um comportamento que exija comunicação em grupo.

Este capítulo descreve o mecanismo de comunicação em grupo proposto para o ambiente de programação paralela/distribuída CRUX [CORSO (1993)].

Em uma tarefa paralela disparada no ambiente paralelo/distribuído, os processos que compõem esta tarefa serão alocados em diferentes nós deste ambiente e executados paralelamente, sendo que a comunicação/sincronização entre os processos será feita através de troca de mensagens.

4.1 Ambiente Paralelo/Distribuído CRUX

O ambiente paralelo/distribuído CRUX é uma máquina paralela baseada em multicomputador. Um conjunto de nós processadores constituem a máquina paralela e são conectados entre si através de dois tipos de dispositivos distintos: um comutador de conexões do tipo *crossbar* e um barramento de serviços como mostrado na Figura 4.1 [MONTEZ (1995)].

As características do modelo de multicomputador considerado são:

- um número expressivo de nós (16 a 64);
- um número moderado de canais por nó (4, por exemplo);

- um comutador de conexões de grande dimensão (de 64x64 a 256x256);
- um barramento de serviço de alto desempenho.

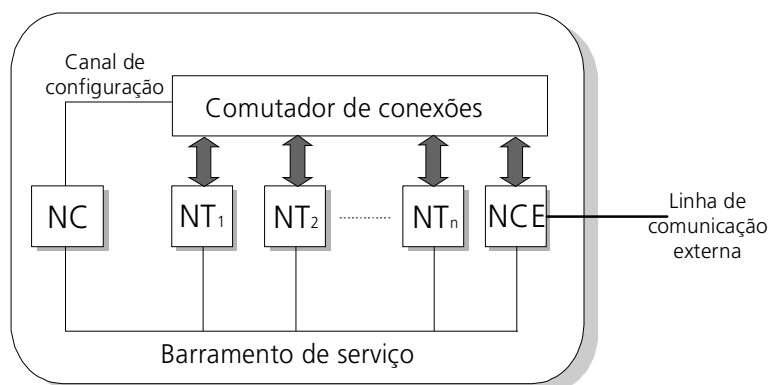


Figura 4.1 - A arquitetura do CRUX

Onde NT_n são os nós de trabalho, NC é o nó de controle e NCE é o nó de comunicação externa, que serão explicados na Figura 4.2 [MONTEZ (1995)].

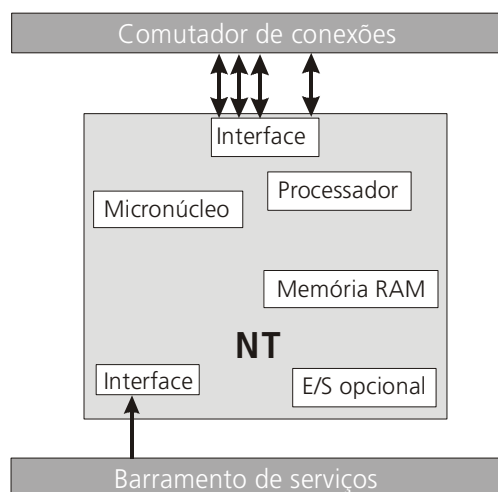


Figura 4.2 – Estrutura interna do nó de trabalho

Cada um dos nós possui um processador, memória RAM e micronúcleo (*microkernel*) com funções básicas que permitem a comunicação entre nós. Além disso, cada nó possui uma ou mais ligações (até quatro) com o comutador de

conexões. O barramento de serviço funciona como uma outra ligação utilizada para a comunicação e controle dos nós.

O comutador de conexões estabelece uma rede de interconexão dinâmica entre os nós processadores. Assim, cada nó pode ser conectado a qualquer outro nó dinamicamente através de canais de comunicação.

O NCE (nó de comunicação externa) é um nó especial neste ambiente e possui a tarefa de fazer a comunicação com o mundo externo através de uma linha de comunicação. O NCE (nó de comunicação externa) é um nó especial neste ambiente e possui a tarefa de fazer a comunicação com o mundo externo através de uma linha de comunicação.

Para trocar mensagens através destes canais, os processos colocados em nós distintos precisam requisitar conexões ao comutador de conexões. Existe um dispositivo inteligente, denominado nó de controle (NC) que é responsável por comandar o comutador de conexões através do canal de configuração. Todos os demais nós de trabalho (NT) podem requisitar conexões bidirecionais, se comunicando com o NC através do barramento de serviços, que funciona como meio de comunicação entre um nó qualquer e o NC. O NC é dedicado exclusivamente a tarefas relacionadas com as funções de gerenciamento dos pedidos de conexões e alocação de nós.

O barramento de serviço é altamente confiável, compartilhado por todos os nós processadores, que servirá para a transmissão de pequenas mensagens entre o NC e os NTs, sendo de uso tão somente do *software* de controle da máquina, com o objetivo de transportar requisições de serviço dos NTs ao NC. Desse modo, este barramento não servirá para comunicação entre NTs, que deverão interagir por meio do crossbar.

A utilização do barramento de serviço para envio e recepção de mensagens de controle é controlada pela camada de Gerenciamento do Barramento de

Serviço (GBS). A camada GBS oferece uma primitiva, *BS_SendRec(msg)*, que envia uma mensagem para o NC e aguarda a resposta.

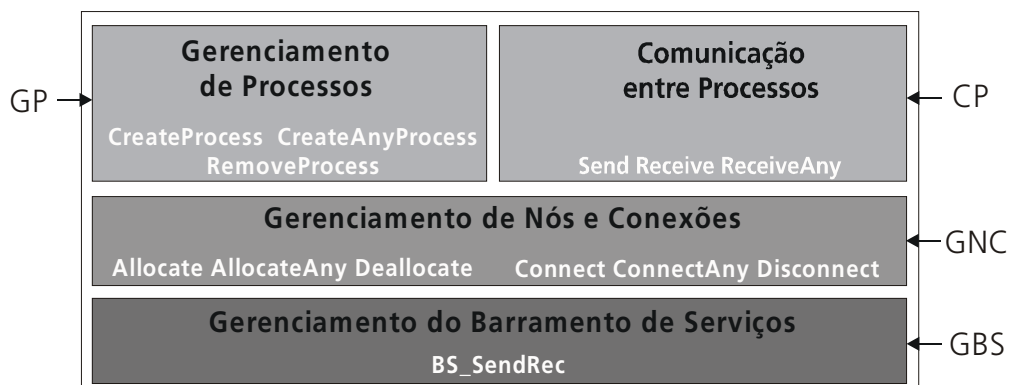


Figura 4.3 - Camadas do micronúcleo do CRUX [MONTEZ (1995)]

Os serviços oferecidos para a utilização do ambiente constituem o micronúcleo do sistema operacional do ambiente [MONTEZ (1995)]. O micronúcleo é composto por camadas conforme mostra a Figura 4.3.

A interface de programação oferecida aos processos de usuário, basicamente, permite a utilização do ambiente em termos de comunicação entre processos e criação de processos.

Estes serviços são oferecidos através das camadas de Gerenciamento de Processos (GP) e de Comunicação entre Processos (CP). O gerenciamento de processos é suportado a partir das primitivas apresentadas na Tabela 4.1.

MENSAGENS	PARÂMETRO	RETORNA	DESCRIÇÃO
CreateProcess	nid		Cria um processo no nó <i>nid</i>
CreateAnyProcess		nid	Cria um processo em qualquer nó
RemoveProcess			Remove o processo do nó atual

Tabela 4.1 – Primitivas de gerenciamento de processos

A Tabela 4.2 descreve as primitivas de comunicação entre processos da camada CP.

MENSAGENS	PARÂMETRO	RETORNA	DESCRIÇÃO
Send	piddes, msg, len		Envia uma mensagem <i>msg</i> de tamanho <i>len</i> para um processo <i>piddes</i> .
Receive	pidsrc, msg, len		Recebe uma mensagem <i>msg</i> de tamanho <i>len</i> originária de um processo <i>pidsrc</i> .
ReceiveAny	msg, len	pidsrc	Recebe uma mensagem <i>msg</i> de tamanho <i>len</i> originária de um processo qualquer cujo pid retorna em <i>pidsrc</i> .

Tabela 4.2 – Primitivas de comunicação entre processos

A implementação das primitivas acima citadas utiliza como suporte as primitivas oferecidas pela camada de gerenciamento de nós e conexões (GNC), estas primitivas são mostradas na Tabela 4.3.

MENSAGENS	PARÂMETRO	RETORNA	DESCRIÇÃO
Connect	nid		Pedido de conexão do nó que solicitou e o nó <i>nid</i>
ConnectAny		nid	Pedido de conexão do nó que solicitou e qualquer nó que deseja se comunicar
Disconnect	nid		Pedido de desconexão entre o nó <i>nid</i> e o nó do processo que pediu a desconexão
Allocate	nid		Pedido de reserva do nó <i>nid</i> para o processo que enviou
AllocateAny		nid	Pedido de reserva de qualquer nó livre
Deallocate	nid		Pedido de liberação do nó do processo que enviou a mensagem

Tabela 4.3 – Primitivas de controle do NC

No ambiente paralelo/distribuído CRUX, em determinado instante, qualquer nó pode ser conectado diretamente a outro através do *crossbar*, eliminando tráfego com roteamento e administração de espaço para armazenamento de mensagens em nós intermediários. Como não há armazenamento temporário de mensagens no nó, não há necessidade de limitar o tamanho das mensagens. A comunicação proposta originalmente é ponto-a-ponto e síncrona, ou seja, um nó envia uma mensagem para um outro nó que está esperando pela mesma.

4.2 Comunicação em Grupo no

Este ambiente pode apresentar um desempenho maior se existir um mecanismo de comunicação em grupo disponível para a comunicação entre os processos que executam nos nós do multicomputador.

4.2.1 Arquitetura do Mecanismo

Com a proposta de fornecer comunicação em grupo pretende-se que os processos possam interagir entre si através do envio e recebimento de mensagens, realizadas por primitivas de comunicação do tipo *GcSend* e *GcReceive*. Neste caso, uma nova camada chamada de Comunicação em Grupo (CG) será adicionada ao micronúcleo do ambiente, como mostra a Figura 4.4. Esta camada será responsável pelo fornecimento da comunicação em grupo. A camada GNC é modificada para fornecer o suporte necessário para o gerenciamento dos grupos e passa a ser chamada de Camada de Gerenciamento de Nós, Conexões e Grupos (GNCG).



Figura 4.4 - Camadas do micronúcleo com comunicação em grupo

Considerando o propósito do ambiente paralelo/distribuído CRUX, os grupos, na maioria das vezes, são fechados. Entretanto, algumas aplicações podem requerer a utilização de grupos abertos. Com o propósito de atender a

todo tipo de aplicação, o sistema proposto suporta tanto grupos fechados como grupos abertos. Os grupos podem ser organizados em pares ou exibindo uma hierarquia, dependendo do tipo de aplicação paralela/distribuída. O endereçamento dos grupos é feito a partir da utilização do identificador do grupo.

4.2.2 Suporte à Comunicação

Conforme limitações da arquitetura hoje, visto que o ambiente CRUX só fornece a possibilidade de comunicação ponto-a-ponto (*unicast*), o suporte de comunicação em grupo deve fazer o envio das mensagens de um grupo através do envio ponto-a-ponto para cada um dos componentes do grupo. A comunicação é síncrona considerando que a maioria das aplicações paralelas/distribuídas utiliza a comunicação também como forma de sincronismo.

4.2.3 Gerenciamento de Grupos

Quanto ao gerenciamento dos grupos, considera-se como solução a implementação totalmente centralizada com um servidor de grupos (SG), que é executado pelo nó de controle (NC). Este servidor mantém um registro de todos os grupos existentes no sistema. Toda vez que um novo grupo é criado, uma requisição deve ser enviada ao servidor de grupos informando a criação deste novo grupo. Cabe a este servidor manter a composição do grupo, que inclui: o identificador do grupo, o tipo de grupo (aberto ou fechado), o tamanho do grupo (estático ou dinâmico), o número de membros que compõe o grupo, os identificadores dos processos que fazem parte do grupo (*pid*) e a localização dos mesmos (*nid*).

Operações de entrada e saída de processos do grupo, também envolvem interação com o servidor. Este tipo de solução apesar de ser simples de ser

implementada, pode tornar o servidor de grupos um gargalo do sistema, principalmente se houver um grande número de grupos a serem gerenciados.

O nó de controle, agora chamado de servidor de conexões está disponível no ambiente CRUX, razão pela qual optamos pela solução centralizada, sendo assim espera-se que esta solução apresente um desempenho plenamente aceitável.

CAMPOS	DESCRIÇÃO
gid	Identificador do grupo
gtype	Tipo do grupo: aberto ou fechado
gsize	Número máximo de membros: estático ou dinâmico
nmembers	Número de processos do grupo no instante
members	Lista dos membros do grupo (nid, pid)

Tabela 4.4 – Base de dados do servidor SG

O servidor de grupos mantém uma base de dados completa de todos os grupos e seus membros. A estrutura da base é uma tabela que armazena informações de cada grupo. Conforme descrito em Tabela 4.4.

A tabela mantida pelo servidor de grupos está ilustrada na Figura 4.5:

gid	gtype	gsize	nmembers	members[i]
0	FECHADO	10	0	{(-1,-1),...}
1	ABERTO	5	2	{(4,345),(1,346),...}
2	ABERTO	0	0	{(-1,-1),...}
3	FECHADO	8	3	{(1,45),(3,47),(5,50)...}

Figura 4.5 – Tabela de grupos do servidor SG

4.2.4 Ordenação e Entrega de Mensagens

É necessária uma semântica bem definida em relação à ordem de entrega das mensagens para tornar a programação clara. A melhor garantia é entregar todas as mensagens instantaneamente na ordem que elas foram enviadas (tempo global de ordenamento [TANENBAUM (1995)]). Entretanto, este tipo de ordenação

nem sempre é implementável. No CRUX existe um relógio privativo para cada nó, o que tornaria necessária a implementação da sincronização dos relógios dos nós, no caso da utilização deste tipo de ordenação.

Na implementação do mecanismo de CG no CRUX as opções foram:

Entrega de Mensagens (atomicidade) - as mensagens de grupo só são válidas quando todos os processos pertencentes a um grupo recebem a mensagem, só assim a operação *GcSend* é completada com sucesso.

Ordenação - é utilizada a idéia de ordenação consistente [TANENBAUM (1995)], onde mesmo que as mensagens são enviadas ao mesmo tempo o servidor de grupo do sistema determina uma como sendo a primeira e realiza a entrega na ordem por ele especificada. Assim é garantido que as mensagens chegam aos membros do grupo na mesma ordem.

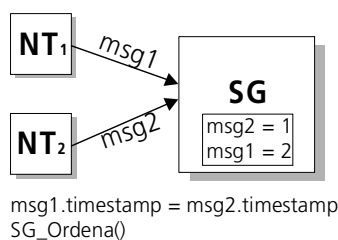


Figura 4.6 – Ordenação consistente através do servidor de grupo

4.2.5 Primitivas do Mecanismo de Comunicação em Grupo

A implementação do mecanismo de comunicação em grupo para o ambiente multicomputador paralelo/distribuído CRUX foi dividida em duas camadas como visto na Figura 4.4, a camada de comunicação em grupo (CG) e a camada de gerenciamento de nós, conexões e grupos (GNCG).

Na camada de gerenciamento de nós, conexões e grupos, as seguintes primitivas estão implementadas (Tabela 4.5). As primitivas de suporte são executadas pelo nó de controle e implementam os serviços de gerenciamento dos grupos e envio/recepção de mensagens.

MENSAGENS	PARÂMETRO	RETORNA	DESCRIÇÃO
CreateG	gtype, gsize	gid	Pedido de criação de grupo de tamanho gsize
DestroyG	gid	ack	Pedido de liberação do grupo gid
JoinG	gid	pos	Pedido de entrada no grupo gid
LeaveG	gid	ack	Pedido de saída do grupo gid
PosInG	gid	pos	Solicita sua posição no grupo gid
SizeOfG	gid	gsize	Solicita o tamanho do grupo gid
ConnectG	nid, pos		Pedido de conexão entre o nó que pediu e o nó pos

Tabela 4.5 - Primitivas da camada GNCG

As primitivas propostas para a camada de comunicação em grupo (CG) são resumidas na Tabela 4.6 e descritas a seguir. Estão divididas em dois grupos: primitivas de gerenciamento (responsáveis pelo gerenciamento dos grupos) e primitivas de comunicação (responsável pelo envio e recepção das mensagens). Estas primitivas representam a interface de programação (API) oferecida para a aplicação. As aplicações chamam estas primitivas, sendo que são as primitivas da camada CG que requisitam (através de envio da mensagem de controle para o NC), os serviços do servidor de grupos.

PRIMITIVAS	PARÂMETROS	
	Entrada	Saída
Gerenciamento		
GcCreate	int gtype, int gsize	int gid
GcDestroy	int gid	int ACK
GcJoin	int gid	int pos
GcLeave	int gid	int ACK
GcPos	int gid	int pos
GcSize	int gid	int size
Comunicação	Entrada	Saída
GcSend	int gid_src, int gid_dst, int pos, char *msg, int len	int ACK
GcReceive	int gid_src, int gid_dst, int pos, char *msg, int len	int ACK

Tabela 4.6 - Primitivas da camada CG

A descrição das primitivas da camada CG, a seguir, envolve também a interação com os serviços oferecidos com a camada GNCG.

int GcCreate(int gtype, int gsize) - Cria um novo grupo de processos. Esta primitiva tem como parâmetro de entrada o tipo do grupo: se for um grupo aberto, *gtype=ABERTO=0*; se for grupo fechado, *gtype=FECHADO=1*; e o outro parâmetro é o número de processos que o grupo comporta. Caso *gsize seja zero*, este é um grupo dinâmico, sem número máximo de processos e se *gsize for diferente de zero*, é um grupo estático. A chamada da função `GcCreate(int gtype, int gsize)` retorna o identificador do grupo a ele associado.

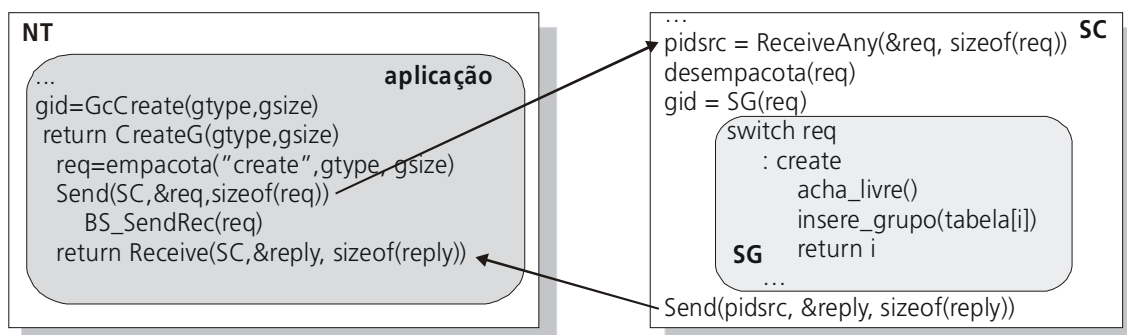


Figura 4.7 – Funcionamento da primitiva GcCreate

Funcionamento:

- o nó de trabalho requisita a criação de um novo grupo de processos através do envio da mensagem `CreateG(gtype, gsize)` para o servidor de grupos do nó de controle;
- o servidor de grupos cria uma nova entrada na tabela e retorna o identificador do grupo (*gid*);
- a entrada na tabela relativa a cada grupo deve conter: identificador do grupo, o tipo de grupo, tamanho do grupo, o número de membros, o *nid* (identificador do nó) e o *pid* (identificador do processo) dos membros do grupo.

int GcDestroy(int gid) - Destrói um grupo de processos já existente. Esta primitiva tem como entrada o *gid* do grupo. Um grupo pode ser destruído implicitamente, quando todos os processos terminaram de executar (grupo estático) ou deixaram de pertencer ao grupo (grupo dinâmico). Ou explicitamente, onde todas as referências ao grupo serão invalidadas na tabela.

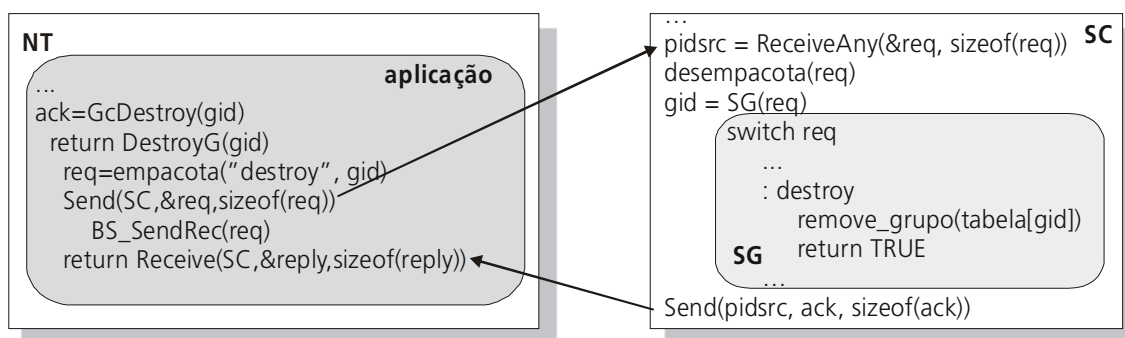


Figura 4.8 – Funcionamento da primitiva GcDestroy

Funcionamento:

- o nó de trabalho requisita o término de um grupo de processos através do envio da mensagem DestroyG(gid) para o servidor de grupos localizado no nó de controle;
- o servidor de grupos verifica se o grupo é estático ou dinâmico. Se for dinâmico, destrói a entrada na tabela, retornando um TRUE como ACK, caso contrário, retorna um erro.

int GcJoin(int gid) - Adiciona ao grupo o processo do nó solicitante. Esta primitiva necessita do identificador do grupo como parâmetro de entrada para adicionar o processo e o nó solicitante na tabela de grupos. Independente do tipo do grupo, dinâmico ou estático, os processos para serem membros de um grupo devem executar esta chamada. O processo-solicitante informa o identificador do grupo

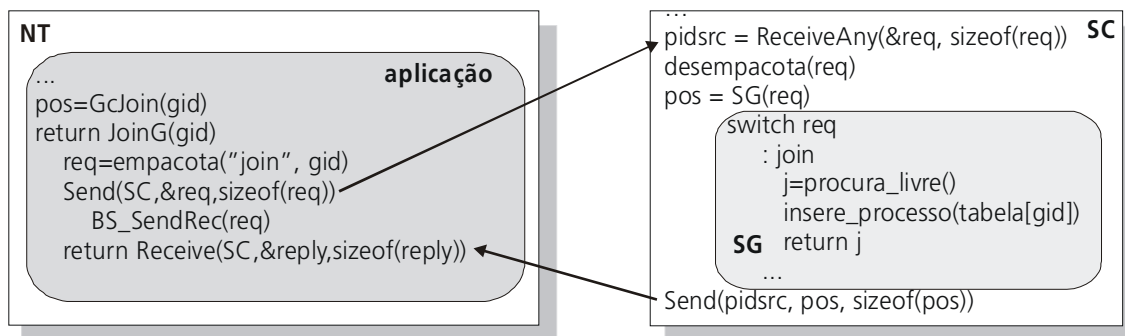


Figura 4.9 – Funcionamento da primitiva GcJoin

Funcionamento:

- o nó de trabalho requisita a inclusão de um processo a um grupo de processos através do envio do pedido para o servidor de grupos do nó de controle, `JoinG(gid)`;
- o processo solicitante juntamente com o seu `nid` é adicionado a lista de membros do grupo cujo `gid` ele solicitou;
- esta primitiva retorna a posição do processo no grupo.

`int GcLeave(int gid)` - Procedimento permitido apenas para grupos dinâmicos, aqueles cujo número de membros é desconhecido (na implementação, é definido como sendo igual a zero). Grupos estáticos não permitem a saída de um processo do grupo, neste caso o processo deixa o grupo somente ao terminar de executar. Esta primitiva tem como entrada o *gid* do grupo. Na saída de um processo, a posição do mesmo dentro do grupo não é mais utilizada, ou seja, a identificação é única.

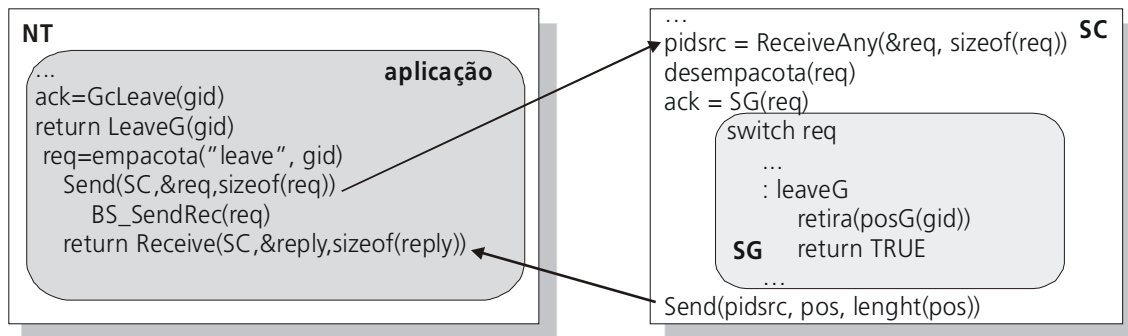


Figura 4.10 – Funcionamento da primitiva GcLeave

Funcionamento:

- o nó de trabalho requisita a retirada de um processo de um grupo de processos através do envio de mensagem para o servidor de grupos do nó de controle, `LeaveG(gid)`;
- o processo solicitante é retirado da lista de membros da tabela de grupos do servidor de grupos no nó de controle; porém sua posição não é reaproveitada, para evitar inconsistências na comunicação entre processos;
- se o membro que está deixando o grupo está bloqueando o grupo pelo fato de não receber alguma mensagem, o grupo é desbloqueado.

int GcPos(int gid) - Indica a posição que o processo ocupa no grupo. Esta primitiva necessita do identificador do grupo ao qual o processo pertence e utiliza `PosInG(gid)`, enviada para o servidor de grupos do nó de controle.

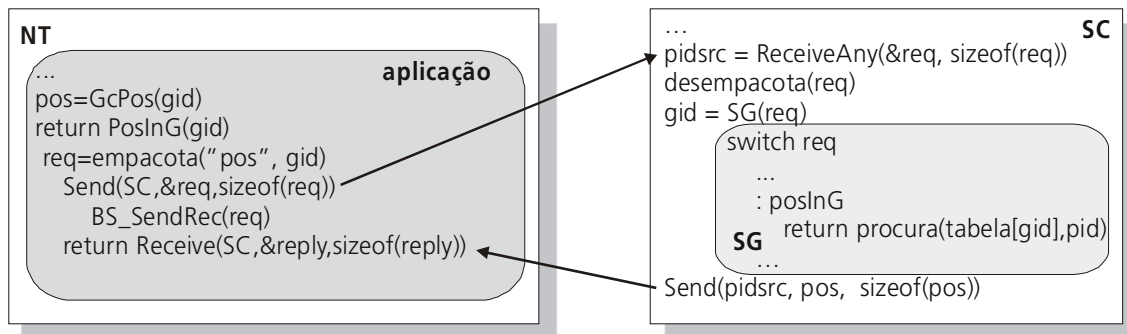


Figura 4.11 – Funcionamento da primitiva GcPos

Funcionamento:

- o nó de trabalho requisita a posição de um processo no grupo de processos através do envio de mensagem para o servidor de grupos do nó de controle, PosInG(gid);
- o nó de controle acha o processo na lista de membros do grupo cujo identificador foi solicitado, retornando a posição do mesmo para quem solicitou.

int GcSize(int gid) – Esta função retorna o tamanho do grupo. Esta primitiva necessita do identificador do grupo ao qual o processo pertence. Envia a mensagem *SizeOfG(gid)* para o servidor de grupos do nó de controle.

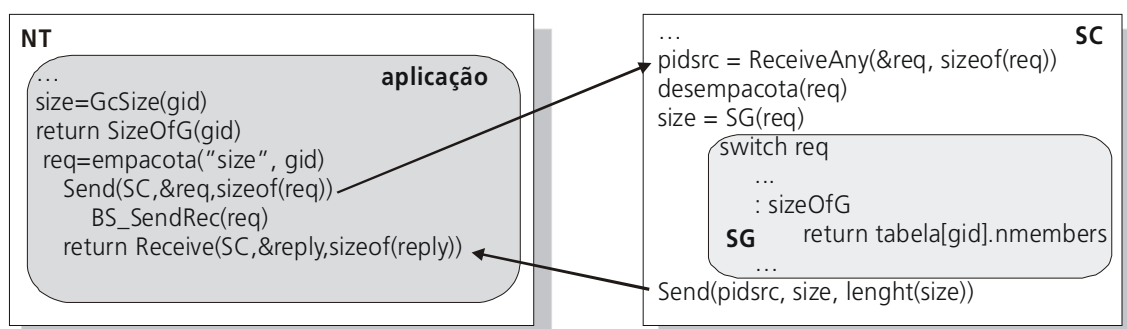


Figura 4.12 – Funcionamento da primitiva GcSize

Funcionamento:

- o nó de trabalho requisita o tamanho do grupo de processos através do envio de mensagem para o servidor de grupos do nó de controle, `SizeOfG(gid)`;
- o nó de controle retorna o valor de `nmembers` como resposta.

`int GcSend(int gid_src, int gid_dst, int pos, char *msg, int len)` - Primitiva de comunicação e envio de mensagens para um grupo. O emissor pode ser um processo não pertencente ao grupo, desde que o grupo seja aberto. Senão os processos não pertencentes ao grupo não poderão enviar mensagens para o grupo. Esta primitiva é usada para o envio síncrono de mensagens, ou seja, todos os membros do grupo devem receber a mensagem para que a operação seja completada com sucesso e o controle retornado para o emissor. Através do parâmetro *pos* é possível enviar uma mensagem para um processo específico do grupo ou para todo o grupo. Se o grupo é estático, esta primitiva é executada somente quando o grupo estiver completo, ou seja, após todos os membros terem entrado no grupo. Quando um processo de um grupo estático termina de executar, o envio de mensagens para aquele grupo (processos que ainda estão ativos) continua normal, embora o número de membros do grupo não seja o declarado na criação do grupo.

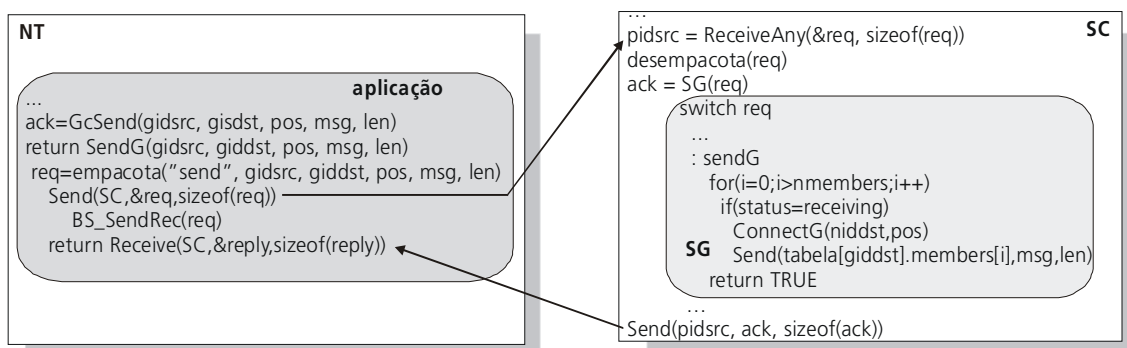


Figura 4.13 – Funcionamento da primitiva GcSend

Funcionamento:

- o nó de trabalho requisita o envio de uma mensagem para um ou mais processos de um grupo de processos através do envio do pedido `ConnectG(gid,pos)` para o servidor de grupos do nó de controle;
- o servidor de grupos do nó de controle estabelece a conexão do nó emissor com qualquer um dos membros do grupo (nós) para envio da mensagem;
- para cada conexão estabelecida existe uma fase de transferência de mensagens;
- quando todos os membros da lista receberem a mensagem, a função retorna com indicação de sucesso.

`int GcReceive(int gid src, int gid dst, int pos, char *msg, int len)` – Primitiva de recepção de mensagens enviadas para um grupo. Esta primitiva bloqueia o processo chamador até que a mensagem especificada seja recebida. As mensagens podem ser recebidas de qualquer grupo ou de um grupo específico. É possível especificar o emissor da mensagem através do parâmetro *pos*.

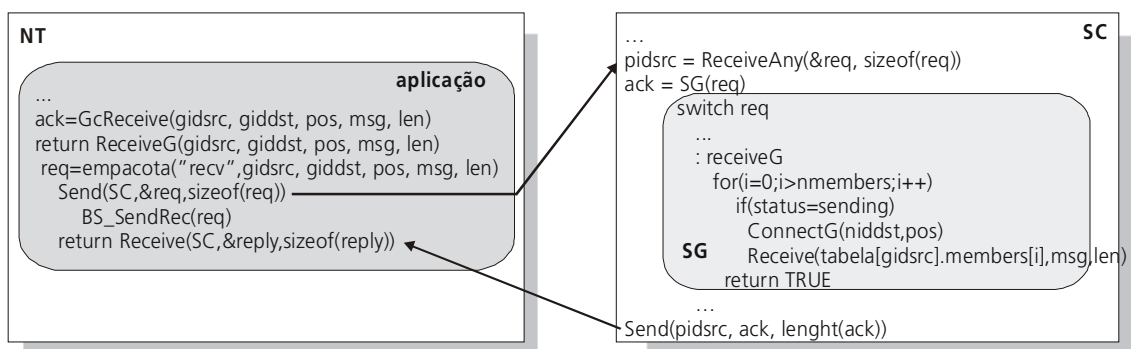



Figura 4.14 – Funcionamento da primitiva GcReceive

Funcionamento:

- o nó de trabalho requisita a recepção de uma mensagem de um grupo de processos através do envio do pedido ConnectG(gid,pos) para o servidor de grupos do nó de controle;
- o servidor de grupos do nó de controle registra a requisição de recepção de mensagem pelo nó solicitante;
- o nó solicitante fica bloqueado até que uma requisição de envio de mensagem é recebida;
- na chegada de uma requisição de envio, a conexão entre o nó receptor e o nó emissor é estabelecida e a mensagem é transferida;
- o controle é retornado somente após a recepção da mensagem.

5 IMPLEMENTAÇÃO DO PROTÓTIPO

O protótipo do mecanismo de comunicação em grupo  foi desenvolvido no Laboratório de Computação Paralela Distribuída – LacPad, no Departamento de Informática e Estatística e no Laboratório de Mecânica de Precisão – LMP, no Departamento de Engenharia Mecânica, ambos no Centro Tecnológico da Universidade Federal de Santa Catarina. O protótipo foi desenvolvido em linguagem C, utilizando o compilador GCC da GNU, em um computador Pentium III com processador 450 MHz e 128Mb de memória RAM.

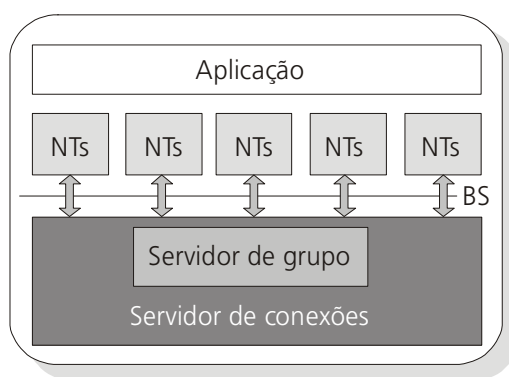


Figura 5.1 – Visão geral do ambiente de comunicação em grupo

O protótipo adotado é um mecanismo simples de comunicação entre nós de trabalho e o servidor de grupos localizado no servidor de conexões.

Quando um processo deseja comunicar com um grupo, ele envia uma mensagem ao servidor de conexões solicitando uma ação. O servidor de conexões por sua vez, verifica com o servidor de grupos quem são os processos e onde eles se encontram. Então estabelece conexões em pares com os membros para qual o processo deseja enviar a mensagem. Cada conexão é seguida por uma etapa de transferência de dados.

5.1 Descrição do protótipo

O protótipo simula a arquitetura representada na Figura 5.1 na plataforma Linux com o objetivo de verificar o comportamento da solução proposta neste trabalho. As funções propostas foram implementadas e descritas de forma a facilitar a migração para o ambiente paralelo/distribuído CRUX.

Os componentes do protótipo são: os NTs, representados por processos do Linux, que são responsáveis pelas chamadas de funções do mecanismo de comunicação em grupo; e o servidor de grupo, representado por um processo do Linux, que é responsável pelo atendimento das requisições dos NTs.

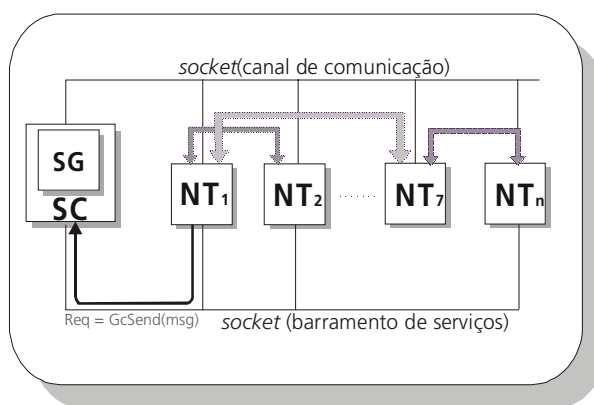


Figura 5.2 - Protótipo do mecanismo de comunicação em grupo

O servidor de grupo é responsável pelo gerenciamento dos grupos assim como pelo envio/recepção da mensagem de/para os grupos (Figura 5.3). Os serviços do servidor de conexões (originalmente nó de controle) são utilizados pelo servidor de grupo para atender as requisições, conectando/desconectando os nós comunicantes.

As requisições de serviço são enviadas para o servidor de conexões através do canal de comunicação que representa o barramento de serviço.

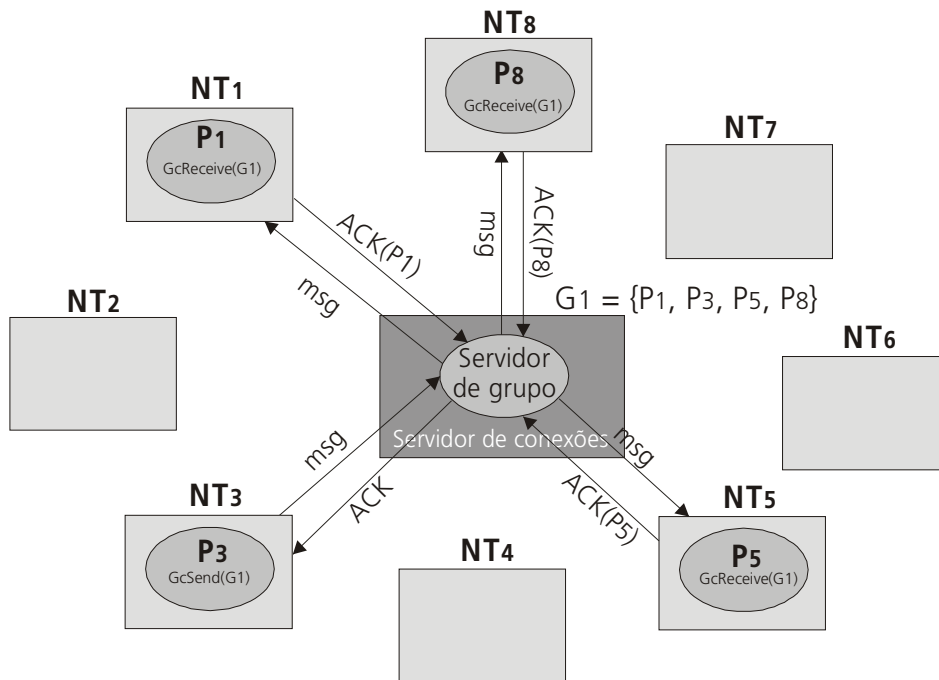


Figura 5.3 - Servidor de grupos

5.2 Servidor de conexões

O servidor de conexões é o protótipo-servidor e funciona da seguinte maneira:

```
void main(void);
{
  InitCRUX();
  InitSC();
  ExitCRUX();
}
```

InitCRUX

Função de inicialização do ambiente CRUX. A partir da chamada desta função, os processos são criados e os canais de comunicação são habilitados.

ExitCRUX

Esta função finaliza o ambiente CRUX no nó. Isto somente ocorre após a verificação de que este procedimento não gerará inconsistências. Após esta verificação, então o ambiente é finalizado.

InitSC

Esta função simula o servidor de conexões do mecanismo. Este servidor é responsável pela inicialização do servidor de grupos, inicialização do barramento de serviços, atendimento das requisições, envio da resposta, como mostra o código abaixo.

```
void InitSC()
{
    char *req;

    InitSG();
    InitBS();
    *req=Espera_requisicao();
    Atende_req(&req);
    Envia_resposta()
}
```

InitBS

Função que inicializa o *socket* que representa o barramento de serviços, abrindo-o para execução que ficará pronto para receber mensagens de requisições dos NTs.

InitSG

Esta função inicializa o servidor de grupos. Recebe uma requisição de operação e utiliza as primitivas de comunicação em grupo para atender estas requisições. As requisições possíveis neste caso são: CreateG, DestroyG, JoinToG, LeaveG, PosInG, SizeOfG.

Nesta função a tabela de grupos é inicializada. Todas as entradas da tabela, correspondente ao número de nós (MAX_NO) que o protótipo está executando, são inicializadas com -1, que define índices vazios na tabela.

O servidor de grupo mantém uma tabela do tipo GTABLE, que consiste dos seguintes componentes:

```
typedef struct tabreg
{
    int gid;
    int gtype;
    int gsize;
    int nmembers;
    t_nid members[MAX_NO];
} GTABLE;
```

onde o tipo t_nid é uma tupla que define os processos-membros do grupo e os nós onde eles rodam, é definido por:

```
typedef struct
{
    int nid;
    int pid;
} t_nid;
```

Espera_requisicao

Esta função utiliza um "ReceiveAny" e espera requisições dos nós de trabalho.

Atende_req (&req)

Verifica qual é a requisição solicitada. Se for requisição para gerenciamento de grupos, envia a requisição ao servidor de grupos, verifica quem são os processos envolvidos, conectando-os para trocar as mensagens.

Envia_resposta ()

Esta função utiliza um "Send" para enviar a resposta para o nó de trabalho.

O protótipo-cliente funciona da seguinte maneira:

```
main(int argc, char*argv[])
{
    int noEnd = TRUE;

    Cria_Socket();
    While(noEnd)
    {
        CriaReq(); //Simula a aplicação criando uma requisição
        EnviaReq(); //Send(Req)
        Recebe(ACK); //Receive(&Resp)
    }
}
```

5.3 Exemplo de Aplicação

Um exemplo de aplicação a ser utilizada é o modelo mestre-escravo, onde o problema a ser resolvido é subdividido em problemas menores, que ficam a cargo de processos trabalhadores, também chamados escravos. O paralelismo é obtido com a execução simultânea dos vários processos escravos.

O mestre é responsável por instanciar os processos escravos nos vários nós do sistema assim como enviar os dados apropriados para cada escravo.

Módulo Mestre:

```
#define MESTRE 0 //Número de mestres
#define ESCRAVOS 5 //Número de escravos
#define ABERTO 0
#define FECHADO 1
#define FALSE 0
```

```
void main()
{
int gid_escravos, gid_mestre, i, e_msg, m_msg, ack=FALSE;

gid_mestre = GcCreate(ABERTO, MESTRE); //Cria um grupo aberto, dinâmico
gid_escravos = GcCreate(FECHADO, ESCRAVOS); //Cria um grupo fechado, estático
m_pos = GCJoin(gid_mestre);
for(i=0;i<=ESCRAVOS;i++)
{
    CreateProcess();
    ChamaPrograma("ES CRAVO");
    m_msg=i;
    ack = GcSend(gid_mestre, gid_escravos,ALL,&m_msg,sizeof(int));
}
if(ack)
for(i=0;i<=ESCRAVOS;i++)
{
    GcReceive(gid_escravos, gid_mestre,ANY,&e_msg,sizeof(int));
    RemoveProcess();
}
}
```

Cada trabalhador é responsável por executar uma tarefa do programa e, após ter realizado o seu trabalho, envia os resultados obtidos para o processo mestre que termina de executar.


Módulo Escravo:

```
void main()
{
int gid_escravos, gid_mestre, m_msg, e_msg, saiu, ack = FALSE;

ack = GcJoin(gid_escravos);
if(ack)
{
    GcReceive(gid_mestre,gid_escravos,m_pos,&m_msg,sizeof(int));
    e_msg = Executa_tarefa(m_msg);
}
```



```
GcSend(gid_escravos, gid_mestre, m_pos, &e_msg, sizeof(int));  
}  
saiu = GcLeave (gid_escravos);  
}
```

Estes são exemplos de aplicação para o protótipo do mecanismo de comunicação .

6 CONCLUSÕES

Mecanismos de comunicação como RPC, são caracterizados por envolverem apenas dois participantes na comunicação, o cliente e o servidor. Entretanto, em ambientes paralelos/distribuídos, existem situações onde a comunicação envolve vários participantes. A utilização de primitivas suportadas por mecanismo de comunicação em grupo, nestes casos, é essencial para um melhor desempenho na comunicação e para facilitar a programação.

Neste trabalho foram realizados: um estudo dos conceitos de comunicação em grupo envolvidos e a descrição de alguns sistemas que suportam a comunicação em grupo. A partir deste estudo juntamente com o levantamento das características do ambiente de programação paralela/distribuída CRUX, foi possível apresentar um conjunto básico de primitivas de suporte à comunicação em grupo para este ambiente. Além disso, foi desenvolvido o protótipo do mecanismo de comunicação cujo objetivo é o de validar as primitivas propostas.

A contribuição deste trabalho está no fato de permitir o desenvolvimento de aplicações que apresentam requisitos de comunicação em grupo no ambiente CRUX, facilitando a expressão deste tipo de comunicação e melhorando o desempenho quando comparado com a comunicação do tipo RPC.

Embora o mecanismo proposto não tenha sido incorporado diretamente no ambiente CRUX, o protótipo desenvolvido permite uma avaliação da funcionalidade das primitivas propostas e prevê uma migração sem muitas adaptações para o ambiente CRUX.

6.1 Perspectivas Futuras

Finalmente, pode-se observar ao longo do desenvolvimento deste trabalho alguns pontos que poderiam ser alvo de futuros desenvolvimentos, objetivando melhorias do trabalho apresentado:

- incorporação do mecanismo proposto no ambiente CRUX e a avaliação de desempenho do mecanismo proposto;
- estudo da possibilidade de tratamento de restrições temporais no mecanismo proposto (suporte a aplicações em tempo real).

7 REFERÊNCIAS BIBLIOGRÁFICAS

[AMIR, DOLEV, KRAMER et al (1992)] AMIR, Y., DOLEV, D., KRAMER, S. et al. *Transis: A Communication Sub-System for High Availability*, 1992 [online]. Disponível na Internet. URL: <<http://www.cs.huji.ac.il/labs/transis/publications.html>> capturado em março 2000.

[AMOEBEA] Amoeba: *User Guide* [online]. Disponível na Internet. URL: <<http://www.cs.vu.nl/pub/amoeba/amoeba.html>> capturado em maio de 1998.

[ANDREWS (1991)] ANDREWS, G. R. *Concurrent Programming: Principles and Practice*. Redwood City: Benjamin/Cummings Publishing Company, 1991.

[BAL, STEINER e TANNEBAUM (1989)] BAL, H. E., STEINER, J. G., TANENBAUM, A. S., *Programming Languages for Distributed Computing Systems*. In ACM Computing Surveys, Vol. 21, p. 261-322, setembro de 1989.

[BIRMAN, JOSEPH e SCHMUCK (1987)] BIRMAN, K., JOSEPH, T. SCHMUCK, F. *ISIS - A Distributed Programming Environment, Version 2.1 - User's Guide and Reference Manual*. Julho de 1987.

[BOKHARI (1981)] BOKHARI, S.H. *On the mapping problem*. *IEEE Transactions on Computers*. New York, v.30, n.3, p.207-214, mar.1981.

[BUYYA (1999)] BUYYA, R., *High Performance Cluster Computing. Architectures and Systems*. Volume 1. Págs 9-22, 1999.

- [CORSO (1993)] CORSO, T. B., Ambiente para Programação Paralela em Multicomputador. Relatório Técnico CPGCC/UFSC n.1, novembro de 1993.
- [COULOURIS, DOLLIMORE e KINDBERG (1994)] COULOURIS, G.; DOLLIMORE, J; KINDBERG, T. *Distributed Systems: concepts and design*. Workingham: Addison-Wesley, 1994.
- [DIJKSTRA (1975)] DIJKSTRA, E. W. Guarded Commands, Nondeterminacy, and Formal Derivation of Programs. *Communications of the ACM*, 18, 8, Agosto de 1975.
- [DOLEV e MALKI (1995)] DOLEV, D., MALKI D. *The Design of the Transis System*, 1995 [online]. Disponível na Internet. URL: <<http://www.cs.huji.ac.il/labs/transis/publications.html>> capturado em março 2000.
- [FOSTER (1995)] FOSTER, I. *Designing and Building Parallel Programs*, 1995. [online]. Disponível na Internet. <URL: <http://www.cs.rdg.ac.uk/dbpp/text/book.html>>.
- [HWANG (1993)] HWANG, K., *Advanced Computer Architecture Programmability*. McGraw-Hill, 1993.
- [KAASHOEK, TANENBAUM e VERSTOEP (1993)] KAASHOEK, M.F.; TANENBAUM, A.S.; VERSTOEP, K. *Group Communication in Amoeba and Its applications*. *Distributed Systems Engineering Journal*, vol 1, pp. 48-58, July 1993.[online]. Disponível na Internet. URL: <http://www.cs.vu.nl/vakgroepen/cs/amoeba_papers.html>
- [KITAJIMA (1995)] KITAJIMA, J.P., Programação Paralela baseado em troca de mensagens. In: Congresso da Sociedade Brasileira de Computação, 15, 1995, Canela/RS. Anais. Porto Alegre: Instituto de Informática da UFRGS, 1995.

- [MACHADO (1997)] MACHADO, R.C.P., GPS: Uma Ferramenta para Programação Paralela baseada em Grupos de Processos, Dissertação de Mestrado - CPGCC da UFRGS - Porto Alegre/RS, 1997.
- [MCBRYAN (1994)] MCBRYAN, O. A., An Overview of Message Passing Enviroments, *Parallel Computing*, p417-444, 1994.
- [MERKLE (1996)] 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, Florianópolis/SC, Fevereiro de 1996.
- [MPI-2 (1996)] MPI-2 Extensions to the Message Passing Interface. *Message Passing Interface Journal*. Outubro de 1996.
- [MONTEZ (1995)] MONTEZ, C. B., Um Sistema Operacional com Micronúcleo Distribuído e um Simulador Multiprogramado de Multicomputador. Dissertação de Mestrado, CPGCC-UFSC, Florianópolis/SC, maio 1995.
- [TANENBAUM, KAASHOEK, RENESSE et al (1991)] Tanenbaum, A.S., Kaashoek, M.F., Renesse, R. van, et al "*The Amoeba Distributed Operating System-A Status Report*" *Computer Communications*, vol. 14, pp. 324-335, *July/August*, 1991 [online]. Disponível na Internet. URL:<
http://www.cs.vu.nl/vakgroepen/cs/amoeba_papers.html>
- [TANENBAUM (1992)] TANENBAUM A. S., *Modern Operating Systems*. Prentice-Hall, 1992.
- [TANENBAUM (1995)] TANENBAUM, A. S., *Distributed Operating Systems*. Prentice-Hall, 1995.