UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# Projeto do Núcleo de um Sistema Operacional Distribuído

por

Benhur de Oliveira Stein

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

Prof. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, outubro de 1992.

# Projeto do Núcleo de um Sistema Operacional Distribuído

por

Benhur de Oliveira Stein

## CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Stein, Benhur de Oliveira

Projeto do Núcleo de um Sistema Operacional Distribuído / Benhur de Oliveira Stein.—Porto Alegre: CPGCC da UFRGS, 1992.

89 p.: il.

Dissertação (mestrado)—Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1992. Orientador: Navaux, Philippe Olivier Alexandre

Dissertação: Sistemas Operacionais, Sistemas Distribuídos, Redes de Computadores
Minix, Comunicação entre Processos

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS

| | |
|---|---|
| CPGCC | Curso de Pós-Graduação em Ciência da Computação |
| DMA | *Direct Memory Access*, Acesso Direto à Memória |
| DUART | *Dual Universal Asynchronous Receiver Transmitter*, Receptor Transmissor Assíncrono Universal |
| FS | *File System*, Sistema de Arquivos |
| FTP | *File Transfer Protocol* |
| IPC | *Inter Process Communication*, Comunicação entre Processos |
| kbyte | quilo byte, aproximadamente mil bytes |
| Mbyte | mega byte, aproximadamente um milhão de bytes |
| MM | *Memory Manager*, Gerente de Memória |
| MMU | *Memory Management Unit*, Unidade de Gerência de Memória |
| NFS | *Network File System* |
| RAM | *Random Access Memory*, Memória de Acesso Direto |
| SOD | Sistema Operacional Distribuído |
| SOR | Sistema Operacional de Rede |
| UFRGS | Universidade Federal do Rio Grande do Sul |

# GLOSSÁRIO

*assembly* linguagem de montagem

*buffer* local de armazenamento temporário

*clock* relógio, sinal que sincroniza as operações de um processador

*flag* variável booleana utilizada para indicar se determinada condição é verdadeira ou falsa

*hardware* conjunto de compunentes físicos de um computador

*link* canal de comunicação

*login* abertura de sessão em um computador

*paging* paginação

*pid* *process identifier*—número que identifica um processo em execução no sistema

*race condition* disputa pelo acesso a um recurso por processos distintos, que pode levar a inconsistências em função da ordem em que esses acessos são realizados

*refresh* re-escrita do conteúdo de memórias dinâmicas

*round-robin* algoritmo de escalonamento no qual os processos são escalonados em ordem seqüencial

*slot* entrada em uma tabela, normalmente a tabela de processos do sistema

*shadowing* artifício para compartilhamento de endereços por processos independentes

*software* conjunto de programas de um computador

*swapping* troca de áreas de memória entre armazenamento primário e secundário

*task* processo responsável pelo tratamento de um dispositivo de E/S

*token* ficha que trafega por uma rede; a estação detentora dessa ficha tem o direito de transmitir, as demais são receptoras

# RESUMO

Uma das tendências para o aumento do desempenho dos sistemas de computação atuais tem sido a distribuição do processamento em uma rede de computadores. Já foram pesquisados diversos modelos para obter essa distribuição, e um dos que tem se mostrado mais promissor é aquele no qual o controle da distribuição é efetuado diretamente pelo sistema operacional. Um sistema operacional desse tipo é chamado de sistema operacional distribuído[TAN85], e seu principal objetivo é fornecer a seus usuários a ilusão de uma máquina uniprocessadora constituída pela soma dos recursos oferecidos pelos componentes da rede. A forma de realizar tal ilusão é o sistema operacional controlar a utilização dos recursos distribuídos para o usuário, independentemente de onde estejam localizados, à medida que sejam requisitados e estejam disponíveis.

Está sendo desenvolvido no CPGCC da UFRGS o projeto DIX, cujo objetivo é o desenvolvimento de um Sistema Operacional Distribuído. Para o desenvolvimento desse projeto, foi tomado como base o sistema operacional MINIX. As principais razões dessa opção foram: o alto grau de modularidade do MINIX, a utilização do paradigma de troca de mensagens para comunicação entre processos e a sua disponibilidade.

A plataforma de *hardware* inicial para o desenvolvimento do projeto é um grupo de estações de trabalho Proceda. Tais estações caracterizam-se por possuir internamente dois elementos processadores distintos. O projeto DIX teve início com o porte do sistema operacional MINIX para o ambiente multiprocessador heterogêneo das estações. Devido à necessidade de comunicação entre as estações e à indisponibilidade de *hardware* adequado para tal, foi desenvolvida uma forma alternativa de comunicação, baseada na utilização da interface paralela existente nas estações.

Este trabalho descreve o núcleo do sistema operacional. A filosofia adotada foi torná-lo o mais simples possível, colocando em processos servidores, externos ao núcleo, grande parte das tarefas. Outro objetivo foi alterar o mínimo possível a interface original do MINIX, para que as camadas superiores do sistema continuassem em funcionamento. Dessa forma, a principal função do núcleo é fornecer aos processos mecanismos para troca de mensagens e transferência de dados entre processos. Foi desenvolvido um método para a identificação global dos processos, que permite identificar cada processo do sistema de forma unívoca e um mecanismo de comunicação entre processos que suporta transparência de localidade, migração de processos e falhas em nodos da rede.

Palavras-chave: Sistemas Operacionais, Sistemas Distribuídos, Redes de Computadores, Comunicação entre Processos

# ABSTRACT

TITLE: "Project of the Kernel of a Distributed Operating System"

One of the modern trends in Computer Science has been the use of distribution to improve system performance. Many models of distribution have been proposed, and the most promising one is that in which the distribution is directly controlled by the operating system. Such type of system is called a distributed operating system[TAN85], and its main goal is to provide its users an illusion of an uniprocessor system more powerful than its components. The operating system controls the utilization of the distributed resources in a transparent way, in order to present such illusion to its users.

There is a project, named DIX, under development at CPGCC/UFRGS, whose goal is to gather experience in the field while developing a distributed operating system. The MINIX operating system has been chosen as a software basis for the project, because of its high degree of modularity, its message passing IPC paradigm and the availability of its source code.

The initial hardware configuration is a set of Proceda workstations. Those workstations have two distincts processors that can run in parallel. The project was started with the porting of MINIX to the workstations' heterogeneous multiprocessor environment. Due to the need of information exchange among the workstations and to the unavailability of suitable communication hardware, an alternative communication scheme was developed.

This work describes the kernel of the operating system. The adopted methodology was to keep it as simple as possible, putting a great number of tasks in server processes outside the kernel. Another goal was to preserve the MINIX original interface, so that the upper layers of the system could remain functional. So, the main purpose of the kernel is to supply an efficient message exchange

mechanism. That mechanism supports locality transparency: the sender of a message is not aware of the destination location, and it is even possible that processes migrate. A method has been developed for the global unique identification of processes.

Key-words: Operating Systems, Distributed Systems, Computer Networks, Inter-Process Communication.

# 1 INTRODUÇÃO

A evolução do hardware e conseqüente queda no custo de sistemas computacionais fez crescer a disponibilidade de estações de trabalho de alto desempenho. Tais estações usualmente são acompanhadas de *hardware* de comunicação de alta velocidade, criando um potencial para processamento cooperativo entre as máquinas.

Apesar de grandes investimentos em pesquisa na área, o *software* não acompanhou o desenvolvimento do *hardware*. O desenvolvimento de *software* para o aproveitamento do potencial deu origem a diversos paradigmas de distribuição: aplicações distribuídas, linguagens distribuídas, bibliotecas de funções, sistemas operacionais de rede, sistemas operacionais distribuídos. Esse paradigmas diferem quanto ao nível de abstração onde a distribuição é implementada.

Dentre esses, o paradigma mais evoluído é o sistema operacional distribuído, pois permite um grau de integração maior que os demais. Um sistema operacional distribuído fornece a seus usuários a ilusão de estarem utilizando uma máquina monoprocessadora virtual. Essa ilusão é obtida através da cooperação entre módulos do sistema distribuído. Tal cooperação requer mecanismos de comunicação eficientes, para que a capacidade computacional dos nodos interligados possa ser aproveitada plenamente.

Está sendo desenvolvido na UFRGS um sistema distribuído experimental, denominado DIX, para uma rede de estações de trabalho Proceda. Estas estações caracterizam-se por apresentarem dois processadores distintos, capazes de operar em paralelo. O sistema operacional fornecido com as estações não aproveita o paralelismo potencial inerente da arquitetura de tais máquinas. O objetivo desse projeto é formar uma base para o estudo de sistemas operacionais distribuídos, bem como obter um melhor aproveitamento do conjunto de estações de trabalho Proceda.

O presente trabalho propõe-se, através do projeto do núcleo do sistema operacional distribuído DIX, estudar e avaliar as questões relacionadas ao desenvolvimento de mecanismos transparentes para o suporte de comunicação e migração de processos. Um protótipo desse núcleo foi implementado, visando validar os mecanismos projetados.

O trabalho está organizado da seguinte forma: no capítulo 2, são apresentados algumas definições, necessárias para o melhor entendimento do restante do presente trabalho. São também discutidos nesse capítulo alguns aspectos relativos a sistemas operacionais distribuídos, em especial os que afetam o projeto do núcleo de um sistema.

O capítulo 3 descreve o projeto DIX, iniciando com a apresentação do sistema MINIX, no qual DIX foi baseado. As características da plataforma de *hardware* empregada são discutidas a seguir. Finalmente, abordam-se os aspectos estruturais do sistema distribuído DIX.

O projeto do núcleo do sistema distribuído proposto é assunto do capítulo 4. São apresentadas as soluções encontradas para a o suporte à comunicação transparente entre processos.

O protótipo desenvolvido para validar os algoritmos apresentados no capítulo 4 é apresentado no capítulo 5. Esse protótipo teve como base o projeto DIX.

Finalmente, no anexo A-1, é apresentado o protocolo de baixo nível implementado para propiciar comunicação de dados entre as estações constituintes da rede.

# 2 SISTEMAS OPERACIONAIS DISTRIBUÍDOS

Cada vez mais espera-se que ambientes computacionais serão compostos por estações de trabalho, interligadas por uma rede local de alta velocidade, com essas redes conectadas por outras de menor desempenho [SHE86]. O objetivo de um sistema operacional distribuído é otimizar o uso dessas redes de computadores.

## 2.1 Conceitos Básicos

Serão apresentados a seguir conceitos básicos e terminologias associadas a sistemas operacionais para redes de computadores. Ênfase especial será dedicada às diferenças existentes entre as duas formas tradicionais de implementação de sistemas para tais ambientes: sistemas operacionais de rede e sistemas operacionais distribuídos.

Não há consenso na literatura quanto aos critérios que distinguem um sistema operacional de rede de um distribuído. Os critérios mais comumente utilizados para essa distinção são: o grau de transparência [TAN85], a heterogeneidade dos componentes [CRI88] e a forma de estruturação interna de cada componente [STA84]. Naturalmente, em alguns aspectos tais critérios estão inter-relacionados. Neste trabalho, o critério considerado mais relevante é o que distingue esses sistemas por meio do grau de transparência apresentado.

### 2.1.1 Sistemas operacionais de rede

Segundo o critério adotado, um sistema operacional de rede (SOR) é um sistema que oferece pouca ou nenhuma transparência a seus usuários. Em tais

sistemas, a utilização de recursos remotos deve ser explicitamente indicada, diferindo sobremaneira do acesso a recursos locais. Tal fato decorre da existência de pouca integração entre os componentes da rede, pois seus sistemas operacionais locais são usualmente distintos.

Um dos principais aspectos a ser analisado, do ponto de vista do usuário, é o mecanismo através do qual arquivos remotos são acessados. A forma mais rudimentar de acesso é exigir que os arquivos sejam copiados da máquina remota para a máquina local para então serem localmente manipulados. Dois exemplos de protocolos utilizados para realizar cópias remotas de arquivos são `uucp` (*unix to unix copy*) e FTP (*File Transfer Protocol*).

Uma maneira mais evoluída é permitir que processos em uma máquina possam acessar arquivos localizados em outra máquina, sem a necessidade de realizar cópias para a máquina local. Nesse caso, o nome do arquivo contém explicitamente sua localização.

Há uma terceira forma, que corresponde à montagem de diretórios remotos na árvore de diretórios local. Nesse caso, o controle da localização dos arquivos passa do usuário comum para o administrador do sistema, embora continue a ser realizada de forma manual. Um exemplo de tal tipo de sistema é o NFS (*Network File System*) [SUN90].

Outro aspecto relevante é a forma como os usuários executam processos remotamente. A primeira abordagem consiste na realização de uma conexão explícita prévia à máquina remota. Nesse caso é necessário que o usuário esteja cadastrado na máquina onde deseja executar seus processos. Comandos usualmente encontrados para efetuar conexões desse tipo são `rlogin` e `telnet`.

Uma outra abordagem, que representa um avanço em relação ao esquema anterior, está baseada em um comando especial, para o qual são fornecidos como parâmetros o nome da aplicação e a máquina onde a mesma será executada.

Neste caso, o ambiente de execução do processo será a máquina remota. `rsh` é um exemplo típico dessa abordagem.

### 2.1.2 Sistemas operacionais distribuídos

Diferentemente de um sistema operacional de rede, um sistema operacional distribuído proporciona um nível de transparência mais elevado. Essa maior transparência é obtida através de uma maior cooperação das máquinas que compõem a rede, que passam a agir de forma integrada.

Em um sistema operacional distribuído, os recursos oferecidos pelas várias máquinas componentes da rede são aproveitados de uma forma global. Através de mecanismos como a migração de processos e de arquivos, é possível compensar o eventual esgotamento de recursos, tais como falta de espaço em disco ou sobrecarga do processador.

Um sistema operacional distribuído oferece a seus usuários a ilusão de uma máquina virtual uniprocessadora, fruto da soma de todos os recursos existentes nos componentes do sistema. Idealmente, a interface apresentada por essa máquina virtual é sempre a mesma, independentemente da localização física do usuário na rede.

Com tal ilusão, o acesso a recursos remotos é realizado de forma idêntica ao acesso a recursos locais. A localização física dos recursos é, inclusive, normalmente escondida do usuário. Isso possibilita que objetos sejam movimentados transparentemente pelo sistema, com o intuito de otimizar o seu desempenho.

## 2.2 Questões de Projeto

Existem várias questões com as quais um projetista de sistema operacional distribuído é defrontado. Aqui serão analisadas os aspectos relacionados com o núcleo do sistema, mais precisamente os destinados a prover transparência de execução. Estes incluem a forma como os processos serão identificados, os mecanismos de comunicação e migração de processos.

### 2.2.1 Identificação de processos

A identificação de processos é um aspecto essencial do sistema. Em um sistema centralizado, a identificação é trivial, enquanto que em um sistema distribuído, deve satisfazer determinados requisitos, como por exemplo:

- identificação global unívoca,
- geração local, por motivos de eficiência,
- independência de localidade.

Várias formas de identificar processos são utilizadas nos sistemas operacionais distribuídos existentes. Algumas delas serão descritas a seguir.

O sistema V [CHE83, CHE86, CHE88] possui um identificador composto pelo nome do nodo onde o processo está executando e uma identificação local única. Este método, apesar de eficiente, não é independente de localidade, pois o nodo onde o processo está localizado faz parte de seu identificador. No sistema DEMOS/MP [POW83], o endereço de um processo é uma tupla <última localização conhecida, <máquina de criação, identificação local>>. A última localização conhecida muda com a localização do processo, enquanto o restante do identificador permanece constante.

O esquema de identificação utilizado no Amoeba [TAN81, MUL86, MUL90, TAN90, MUL87] está baseado em capacidades (*capabilities*). Essas capacidades são utilizadas para identificar todos os tipos de objetos presentes no sistema. A unicidade dos identificadores é proporcionada pela enorme gama de valores assumíveis ($2^{48}$).

### 2.2.2 Comunicação entre processos

A comunicação entre processos está intimamente relacionada com a forma com que os processos são identificados; influencia também os mecanismos de migração de processos. Nos sistemas analisados, a comunicação é realizada através do paradigma de troca de mensagens.

No sistema V, a forma de realizar a comunicação é bastante simples, basta enviar a mensagem ao nodo indicado no identificador do processo destino. Já no Amoeba, o processo que deseja enviar uma mensagem fornece a capacidade que identifica o objeto com quem quer se comunicar. Codificada nessa capacidade está a identidade do processo servidor desse objeto, para onde a mensagem é enviada.

Outra forma de realizar a comunicação entre processos é através de *links*, que podem ser considerados canais para troca de mensagens. São manipulados de forma semelhante a capacidades. O sistema DEMOS/MP [POW83, SMI88] utiliza *links* monodirecionais, ao passo que o sistema Charlotte [ART87, FIN89] usa uma versão bidirecional desse esquema.

### 2.2.3 Migração de processos

Existem vários motivos que tornam desejável a existência de migração de processos em sistemas operacionais distribuídos [SMI88, MUL87], entre os quais podem ser citados:

- Equilibrar a distribuição da carga entre os processadores da rede. Esse equilíbrio pode causar grande impacto no desempenho do sistema.
- Reduzir o tráfego na rede, quando um processo opera sobre um grande volume de dados remotos. Dependendo do tamanho do processo em relação aos dados que utiliza, pode ser vantajoso mover o processo para o nodo que contém os dados, ao invés de mover os dados até o processo.
- Utilizar recursos que não são acessáveis remotamente. Nessa categoria estão dispositivos especiais de *hardware*, tais como um processador vetorial, por exemplo.
- Em alguns sistemas, estações de trabalho pessoais ligadas à rede são utilizadas para conter processos do restante do sistema quando não estão sendo utilizadas por seus donos. Esses processos devem migrar para outros nodos quando o dono da estação retorna à mesma.
- Casos em que é previsto que o processador onde o processo está sendo executado não estará mais disponível. Por exemplo, o processador pode ser desligado ou alocado a um processo de tempo-real especial, dedicado a esse processador.

Apesar de bastante desejável, a migração de processos não é uma tarefa simples. Segundo Tanenbaum [TAN85], é próximo do impossível na prática. A migração pode ser dividida em quatro etapas:

- o processo é suspenso no processador origem
- o estado do processo é transferido ao processador destino
- o processo é reiniciado no processador destino
- o processo continua a acessar os recursos que utilizava, sem que para isso seja necessário esforço extra

Existem dois aspectos que dificultam essa operação, o espalhamento do estado do processo em locais distintos do sistema e a necessidade de um alto grau de transparência que possa garantir a continuidade de acesso aos recursos. O estado de um processo inclui, por exemplo, a posição corrente em arquivos abertos, o valor de temporizadores que o processo disparou, o identificador do processo, sinais pendentes, etc. Essas informações devem possuir no local destino o mesmo conteúdo semântico que possuiam onde o processo estava executando antes de migrar. Adicionalmente, a migração não deve consumir tempo excessivo, para não afetar os processo que interagem com o processo que migra.

Quanto menor e mais simples for o núcleo do sistema, menor será a quantidade de informação que ele conterá sobre os processos, simplificando a implementação de mecanismos que possibilitem a migração de processos.

A migração de processos afeta a comunicação entre processos. A nova localização do processo que migrou deve ser encontrada pelos nodos que desejam lhe enviar mensagens. Em alguns sistemas, por exemplo o DEMOS/MP, o nodo que continha o processo mantém alguma indicação sobre o paradeiro do mesmo. Em outros, tal como no Amoeba, é utilizado um esquema de *broadcast* para localizá-lo.

# 3 DESCRIÇÃO DO PROJETO DIX

Desde 1988, o grupo de sistemas operacionais e o grupo de arquiteturas de computadores do CPGCC desenvolve o projeto M3P [CAR89, BEL90, STE89]. O objetivo desse projeto é o desenvolvimento e implementação de uma máquina multiprocessadora e de seu respectivo sistema operacional. Tal sistema está baseado na adaptação do MINIX (um sistema operacional compatível com o UNIX e cujo código fonte é disponível para o meio acadêmico) às características do multiprocessador. Devido a problemas diversos, a implementação do *hardware* da máquina sofreu atrasos impedindo que o grupo de sistemas operacionais prosseguisse os testes do *software*.

Aproximadamente na mesma época, o CPGCC adquiriu um grupo de estações de trabalho Proceda modelo 5370-CAD, que se caracterizam por possuir um *hardware* de bom desempenho, composto basicamente por dois processadores e um *display* gráfico de alta resolução. O sistema operacional fornecido com as estações é monoprogramado e monoprocessado, não explorando adequadamente os recursos oferecidos pelas mesmas.

Iniciou-se então o projeto DIX [BAR90, BAR90a], cujo objetivo era produzir um sistema operacional capaz de oferecer:

- multiprogramação em ambos os processadores,
- multiprocessamento, com uso intensivo e simultâneo dos processadores de cada estação,
- processamento distribuído, com as estações cooperando de forma a prover transparência de localidade de processamento e de arquivos.

A experiência dos autores com o sistema operacional MINIX, adquirida durante o projeto M3P, somada às características desse sistema, como sua

estruturação em camadas e a disponibilidade de seu código fonte, tornaram natural a escolha do MINIX como ponto de partida para o projeto DIX.

O restante deste capítulo descreve a reestruturação do sistema MINIX realizada no DIX, iniciando por uma descrição sucinta do MINIX e das estações Proceda.

## 3.1 O Sistema Operacional MINIX

O sistema operacional MINIX [TAN87] foi desenvolvido na Universidade Livre de Amsterdam, e colocado à disposição do meio acadêmico para ensino e pesquisa. O sistema foi quase que totalmente escrito na linguagem C [KER78], com poucas rotinas em *assembly*. O MINIX foi desenvolvido para computadores pessoais compatíveis com o IBM PC e é funcionalmente compatível com o UNIX versão 7.

A estrutura interna do MINIX, entretanto, é bastante diferente daquela do UNIX [RIT74]. O UNIX é organizado como um sistema monolítico—é um único programa que contém todas as funções do sistema operacional. O MINIX, por outro lado, é estruturado de uma forma mais modular, como uma coleção de processos independentes que se comunicam uns com os outros e com os processos de usuário através de um mecanismo de troca de mensagens. Esses processos estão estruturados em quatro camadas, conforme mostra a figura 3.1.

### 3.1.1 Camada 1—Gerência de processos e mensagens

Nesta camada estão localizados os mecanismos que implementam a multiprogramação e a comunicação entre processos do sistema. A comunicação entre processos é realizada por meio de troca de mensagens, que são estruturas de

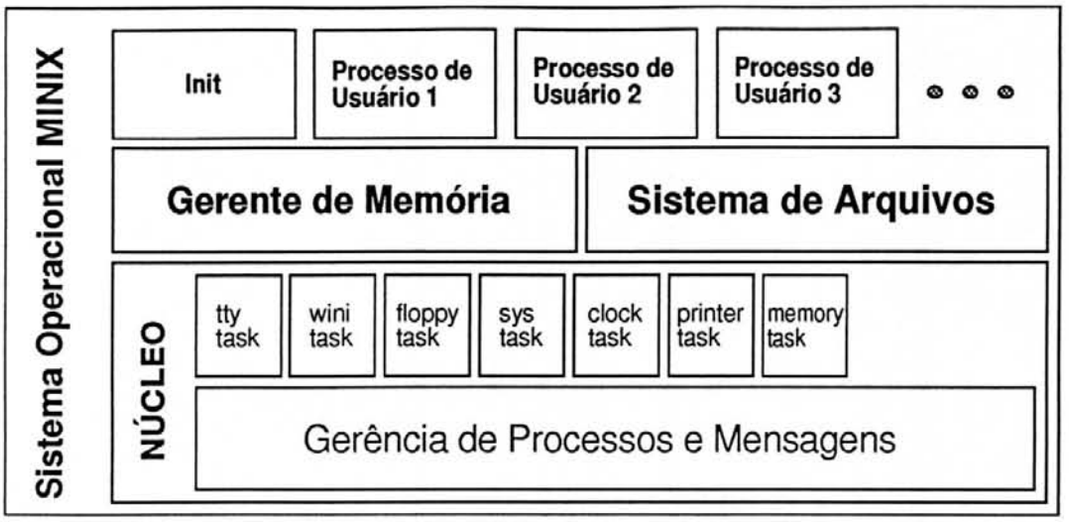


Figura 3.1: Camadas do sistema operacional MINIX

dados de tamanho fixo (24 bytes, no caso de um IBM PC). Essa estrutura contém a identificação do processo que envia a mensagem, o tipo da mensagem e os parâmetros da mesma.

Existem três primitivas para a troca de mensagens, cada uma com dois parâmetros: a identificação do processo com o qual deseja-se efetuar comunicação e o endereço de um *buffer* que contém a mensagem. Essas primitivas são:

**send** envia uma mensagem a outro processo. Se o processo destino estiver aguardando uma mensagem do processo que a envia, a mensagem é copiada para o *buffer* desse processo e ambos são desbloqueados. Caso contrário, o processo enviando a mensagem é bloqueado.

**receive** informa ao sistema que o processo quer receber uma mensagem de um processo específico, ou de ANY, no caso de aceitar qualquer mensagem. Se existir um processo bloqueado em `send` para o processo que fez esta chamada, a mensagem é copiada. Senão, o processo é bloqueado em `receive`.

**sendrec** é uma combinação das outras duas chamadas. Envia uma mensagem ao processo especificado e bloqueia à espera de uma resposta do mesmo.

Essas primitivas são as únicas verdadeiras chamadas de sistema, realizadas através de interrupções de *software* ao núcleo do sistema. A outra forma de entrar-se no núcleo é por meio de interrupções de *hardware*. Neste caso, o núcleo prepara uma mensagem informando o tipo de interrupção e a envia ao processo da camada 2 responsável pelo tratamento de tal tipo de interrupção.

A cada interrupção, seja de *software* ou de *hardware*, o núcleo salva o contexto do processo que estava sendo executado antes de atender a interrupção, para poder retomar sua execução posteriormente. Esse contexto, juntamente com outras informações relativas ao processo, são armazenadas em uma tabela, denominada tabela de processos, que contém uma entrada para cada processo no sistema. A identificação de processos, para envio e recepção de mensagens, é efetuada através de um número que corresponde à entrada do processo nessa tabela. Assim, o processo que ocupa a quinta posição da tabela é identificado com o número 5, e assim sucessivamente. Os processos da camada 2 possuem números negativos, o gerente de memória (MM) é o processo 0, o sistema de arquivos (FS) é o processo 1, o init (processo de inicialização do sistema) é o processo 2 e os processos de usuário são identificados por números a partir de 3. Quando um processo morre, sua entrada na tabela é colocada à disposição, de forma que um outro processo pode receber o mesmo número de um processo que já morreu. Esse fato não representa problema, pois o sistema, antes de liberar a entrada de um processo, cancela possíveis chamadas pendentes para esse processo. Adicionalmente, não é permitida a troca de mensagens diretamente entre processos de usuário.

O escalonamento de processos é realizado por meio de três filas de prioridades diferentes, correspondendo às três camadas do sistema que contêm processos. A fila de maior prioridade contém os processos da camada 2, e a de menor prioridade, os processos de usuário (camada 4). Somente quando não existirem processos prontos para execução em uma fila de maior prioridade é permitido aos processos das filas de menor prioridade disputarem o processador.

Dentro de cada fila, os processos são escalonados segundo o algoritmo de *round-robin*. Os processos de usuário ainda podem perder o processador por tempo. A cada 100 ms, o sistema verifica qual o processo de usuário que está em execução. Se o mesmo processo estiver executando por mais de 100 ms, é colocado no final da fila de processos de usuário e o próximo processo dessa fila é selecionado para execução.

### 3.1.2 Camada 2—Tratadores de dispositivos de E/S

No MINIX, cada dispositivo de E/S é tratado por um processo independente, denominado *task*. Existe uma *task* para cada dispositivo presente no sistema (relógio, terminais, discos, impressora, ...). Cada *task* executa sempre o mesmo laço principal, composto pelas seguintes etapas:

a. espera uma mensagem,

b. efetua a operação requisitada pela mensagem,

c. envia uma mensagem com a resposta da operação.

Além das *tasks* que controlam os dispositivos de E/S, existe uma *task* especial, denominada SYS-TASK. A função dessa *task* é possibilitar a comunicação dos servidores com o núcleo do sistema. É através dessa *task*, por exemplo, que o MM informa o núcleo sobre a criação e morte de processos, altera mapas de alocação de memória ou envia sinais. Uma outra função da SYS-TASK é efetuar cópias de dados entre processos. É dessa forma que os servidores realizam cópias de dados entre os processos de usuário e as *tasks* de E/S, por exemplo.

As camadas 1 e 2 ocupam o mesmo espaço de endereçamento, e são conjuntamente chamadas de núcleo do sistema MINIX. Todos esses processos exe-

cutam em modo supervisor (quando o *hardware* permite), tendo acesso irrestrito aos recursos do sistema, como memória e dispositivos de E/S.

### 3.1.3 Camada 3—Servidores

A camada 3 do MINIX contém dois processos, chamados de servidores: o gerente de memória (MM—Memory Manager) e o servidor de arquivos (FS—File System). Eles são estruturados de forma análoga às *tasks*, ou seja, recebem uma mensagem, executam o que lhes é pedido e enviam uma resposta. Esses servidores são disjuntos do núcleo, executando em modo não privilegiado. Toda a comunicação com o núcleo é realizada exclusivamente por meio de trocas de mensagens com as *tasks*. São os servidores que fornecem todos os serviços do sistema operacional disponíveis aos processos de usuário. Não é permitido aos processos de usuário enviar mensagens diretamente às *tasks*.

O MM processa as chamadas de sistema relacionadas com alocação de memória e gerência de processos. Por exemplo, as chamadas FORK, EXEC, WAIT, KILL, BRK são todas tratadas por esse servidor. O algoritmo de alocação de memória implementado pelo MM é bastante simples, não havendo mecanismos como *swapping* ou *paging*.

O FS implementa um sistema de arquivos semelhante ao encontrado no UNIX. Os pedidos de operações de E/S dos processos de usuário são tratados pelo FS. Ao receber um pedido, o FS identifica o dispositivo que contém o arquivo que o usuário pretende acessar, e efetua as chamadas necessárias à *task* que controla esse dispositivo. Os pedidos dos usuários são tratados pelo FS de forma seqüencial, isto é, nunca há mais de um pedido sendo tratado ao mesmo tempo.

### 3.1.4 Camada 4—Processos de usuário

Nesta camada localiza-se o processo *init*, os utilitários do sistema (interpretador de comandos, editor de textos, compilador, ...) e todos os processos de usuários. O *init* é um processo especial, carregado juntamente com o sistema, que dispara um processo de *login* para cada terminal ligado ao sistema. Os processos situados nesta camada podem enviar mensagens somente ao MM ou ao FS.

## 3.2 As Estações de Trabalho Proceda

As estações de trabalho Proceda 5370-CAD possuem dois processadores heterogêneos, um Motorola 68020 e um Intel 8088. Sua arquitetura é mostrada na figura 3.2.

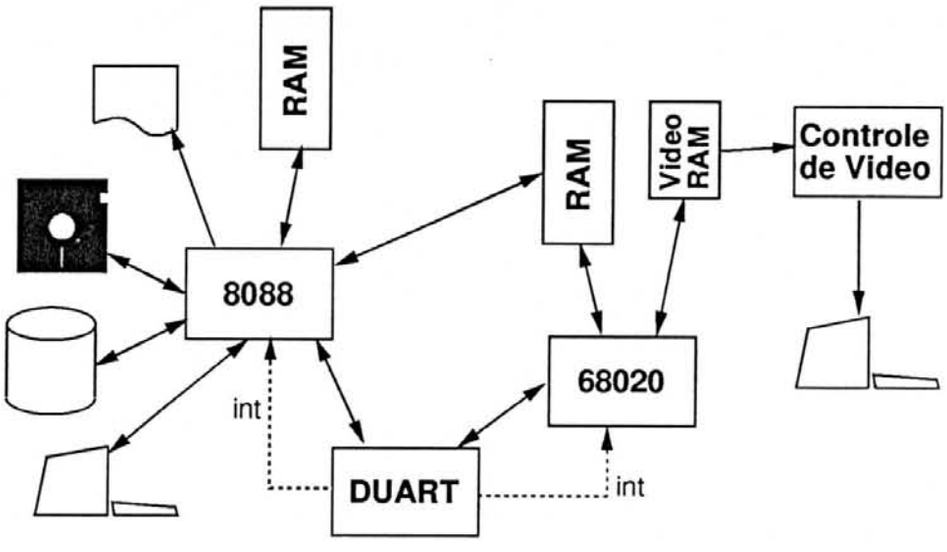


Figura 3.2: Arquitetura da estação de trabalho Proceda 5370-CAD

O processador principal, um Motorola 68020, possui *clock* de 20MHz, e é auxiliado por um coprocessador aritmético Motorola 68881. Esse processador dispõe de 4 Mbytes de memória RAM e 2 Mbytes de memória de vídeo, controlada por um processador gráfico Intel 82786. O 68020 controla ainda uma

interface DUART Motorola 68681 para comunicação serial e interrupção interprocessador. Esses componentes são hospedados por um computador IBM PC-XT convencional.

Os dispositivos de E/S são controlados pelo processador do IBM PC, um Intel 8088. Esses dispositivos são: a console, um disco rígido, uma unidade de disco flexível, duas portas de comunicação serial e uma de comunicação paralela.

Para possibilitar a comunicação entre os processadores, existe uma linha de interrupção do 8088 ligada à DUART, controlada pelo 68020. Adicionalmente, o 8088 tem acesso à memória do 68020. Esse acesso é realizado por meio de uma janela de 64 kbytes, localizada no topo da memória do 8088 e que pode ser mapeada para qualquer endereço da memória do 68020. O controle da DUART é efetuado por meio de acessos à posições especiais da memória do 68020, que são mapeadas para os registradores da DUART. Dessa forma, o 8088 tem acesso à DUART, podendo interromper o 68020 através da mesma.

## 3.3 Porte do MINIX para as Estações Proceda

O passo inicial do projeto DIX foi o porte do sistema operacional MINIX para as estações Proceda. Inicialmente, decidiu-se como seria estruturado o sistema, ou seja, como os módulos do sistema seriam distribuídos entre os dois processadores da estação. Essa divisão está esquematizada na figura 3.3. Os dois processadores contêm rotinas da camada 1, já que ambos são multiprogramados, e os processos existentes em ambos enviam e recebem mensagens. Como os periféricos estão ligados ao 8088, sua tarefa principal é controlar tais dispositivos, operando como um processador satélite [KUT84]. Praticamente todas as *tasks* da camada 2 estão localizadas no 8088. No 68020, existem somente duas *tasks*: a SYS-TASK, que realiza a interface entre o núcleo do sistema e os servidores, e a MEM-TASK, que possibilita o acesso à memória do 68020 como se a dispositivo

de E/S. Os servidores (MM e FS) estão no 68020, a fim de aproveitar o maior poder de processamento desse processador, e também para residirem no mesmo processador onde estão situados os processos de usuário.

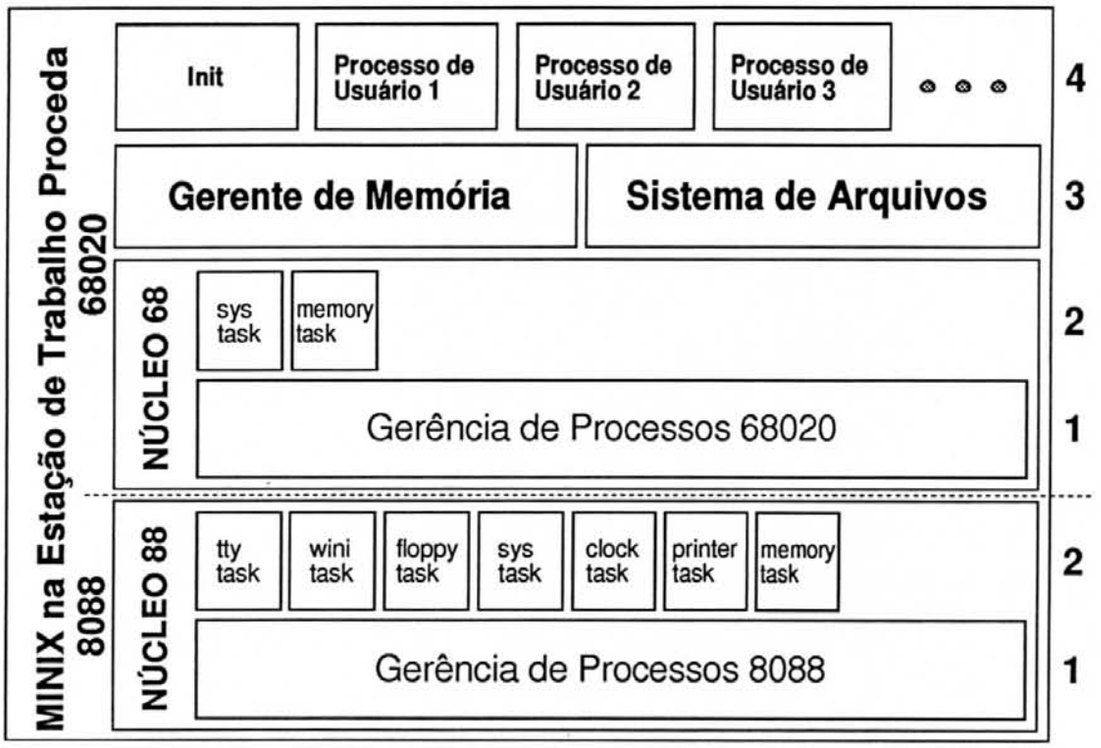


Figura 3.3: Camadas do MINIX nas estações Proceda

Há dois métodos de comunicação entre processos no MINIX: troca de mensagens e transferência de blocos de dados. Como existem processos localizados em ambos os processadores, foi necessário criar mecanismos para possibilitar as duas formas de comunicação quando os processos estão localizados em processadores distintos.

### 3.3.1 Troca de mensagens interprocessadoras

A forma que os processadores têm para comunicarem-se é através da memória do 68020 (que pode ser acessada pelo 8088) e das linhas de interrupção mútua controladas pela DUART.

Para a troca de mensagens entre processadores, foram alocados dois *buffers* em endereços fixos da memória do 68020, um para mensagens oriundas do 8088 para o 68020 e outro para mensagens no sentido inverso. Existe um *flag* associado a cada um desses *buffers*, que informa se o mesmo está livre ou ocupado. O endereço desses *buffers* é conhecido pelo núcleo do 8088. Para enviar uma mensagem ao outro processador, o núcleo do processador origem aguarda até que o *flag* indique que o *buffer* está livre. A seguir, coloca a mensagem no *buffer*, muda o estado do *flag* para ocupado e interrompe o outro processador. Ao ser interrompido, o processador destino verifica o estado do *flag*, que deverá estar ocupado, copia a mensagem localizada no *buffer* compartilhado para um *buffer* interno e coloca o *flag* em estado livre. Não existem problemas de *race conditions*, pois o acesso ao *buffer* compartilhado é controlado por um *flag*, que indica qual dos processadores tem acesso ao *buffer* e ao próprio *flag*. Um processador somente pode alterar o conteúdo dessas estruturas quando o outro processador passar-lhe a vez, alterando o valor do *flag*.

Os dois processadores existentes na estação são bastante diferentes, um possui palavras de 32 bits e o outro de 16; em um deles, a ordem dos bytes em uma palavra é ascendente, enquanto que no outro, descendente. Devido a essas diferenças, é necessário efetuar uma conversão nas mensagens quando elas passam de um processador para outro. Essa conversão é realizada pelo núcleo do 8088, logo que recebe uma mensagem do 68020 ou logo antes de enviar uma mensagem para o mesmo. Analisando o tipo da mensagem e seu destino, é possível determinar quais os tipos dos campos componentes da mensagem. A tabela 3.1 ilustra os tipos utilizados nas mensagens do MINIX e as conversões necessárias.

Nas mensagens destinadas às *tasks* (que são as mensagens interprocessadoras), os campos do tipo `int` são sempre utilizados para armazenar grandezas representáveis em 16 bits. Assim, os 16 bits mais significativos de um inteiro de 32 bits do 68020 são desprezados sem problemas.

Tabela 3.1: Conversões de tipos em mensagens interprocessadoras

| *Tipo* | *Conversão* |
|---|---|
| `char` | sem conversão |
| `char[]` | sem conversão |
| `long` | inverter ordem dos bytes |
| `int` | inverter ordem dos bytes; truncar ou preencher com zeros |
| apontador | inverter ordem dos bytes; ver texto |

Os apontadores são mais complexos: um apontador possui 32 bits no 68020 e contém um valor que pode endereçar qualquer posição na memória desse processador. Seus 16 bits superiores não podem ser desprezados. A solução adotada foi a conversão de tipos em mensagens do 8088 que utilizavam apontadores. Ao invés de apontadores, essas mensagens passam a conter variáveis do tipo `long`, que contêm 32 bits em ambos os processadores.

A função dos apontadores em mensagens é passar endereços para onde serão copiados os dados lidos pelas *tasks* dos dispositivos de E/S (ou de onde serão lidos dados para serem escritos nesses dispositivos). No MINIX, as *tasks* utilizam duas formas diferentes para efetuar cópias de dados de/para os demais processos: por meio de chamadas diretas às funções `get_byte` e `put_byte`, quando a cópia de dados é realizada byte a byte (nas *tasks* que controlam os terminais e a impressora), ou através de chamadas à função `phys_copy`, para cópias de blocos de dados (nas *tasks* de disco e demais periféricos orientados a blocos).

Para simplificar o porte e tornar mais modular o sistema, essas funções não são mais utilizadas. As cópias de dados foram centralizadas na SYS-TASK, que realiza cópias de dados ao receber uma mensagem do tipo `SYS_COPY`. Essa chamada já era utilizada para realizar as cópias de dados para os servidores. Dessa forma, os endereços recebidos pelas *tasks* de E/S não são mais tratados como

endereços pelas mesmas, podendo ser armazenados em variáveis do tipo `long`. Foi necessário alterar todas as *tasks* para solucionar essa questão de representação dos apontadores.

### 3.3.2 Cópias de dados

As cópias de dados entre processos em diferentes processadores são sempre realizadas a partir do comando de uma *task*, localizada no 8088. Por esta razão, e em função de o 8088 ser o único processador que tem acesso à memória de ambos os processadores, as cópias de dados interprocessador são realizadas pelo 8088, mais especificamente pela SYS-TASK. Com todas as cópias de dados centralizadas na chamada `SYS_COPY`, essa tarefa tornou-se mais simples. Uma mensagem do tipo `SYS_COPY` recebe como parâmetros o processo, endereço e segmento (código, dados ou pilha) origem e destino, além do número de bytes a transferir. Analisando-se a identificação do processo, é possível descobrir a qual dos processadores pertence. Como o 8088 tem acesso à memória do 68020, a realização da cópia é simples.

### 3.3.3 *Shadowing* de processos

O processador 68020 permite endereçamento absoluto a posições de memória. No *hardware* das estações Proceda não existe uma MMU (Memory Management Unit), que permitiria tornar esses endereços relativos. Devido a isso, os programas têm de ser relocados durante sua carga para que possam ser executados em qualquer posição da memória do 68020, não podendo ser movidos dessa posição enquanto estiverem sendo executados.

No UNIX, a criação de processos é realizada através da chamada FORK, que duplica toda a imagem de um processo já existente. Após duplicados, os

processos possuem segmentos independentes de código, dados e pilha, que são inicialmente iguais. Como ambos os processos possuem segmentos de código iguais, realizarão acessos a dados colocados nas mesmas posições absolutas de memória. As alterações realizadas por um dos processos em seu segmento de dados não devem se refletir no segmento de dados do outro, e os segmentos devem iniciar na mesma posição absoluta de memória. A forma de resolver esse problema é o algoritmo de *shadowing*, em que é alocada uma área de memória, chamada sombra, onde é colocado o segmento de dados e pilha do processo que não está sendo executado. Cada vez que é trocado o processo em execução, entre os que compartilham endereços de dados, é copiada a área de memória da posição dos dados para a região de sombra e vice-versa.

Quando um desses processos realiza uma operação de E/S, seu segmento de dados é colocado na região de sombra, e é bloqueado nessa região até que a operação se complete. Isso é feito para que o outro processo possa executar enquanto a operação de E/S é realizada. Para que as *tasks* de E/S localizadas no 8088 possam colocar ou retirar os dados do processo correto, é necessário realizar um mapeamento dos endereços enviados nas mensagens interprocessadoras. Caso esses endereços pertençam a um processo que está colocado em uma região de sombra, devem ser alterados para refletir a posição correspondente dentro dessa região. Esse mapeamento é realizado pelo núcleo do 68020, logo antes de enviar a mensagem ao 8088.

### 3.3.4 Escalonamento e contabilização dos processos do 68020

A DUART possui uma linha de interrupção ligada ao 68020, normalmente utilizada para gerar interrupções periódicas para esse processador. No caso do DIX, essa interrupção é utilizada pelo 8088 quando necessita enviar uma mensagem ao 68020, não existindo uma forma de o 68020 controlar diretamente o tempo de execução de seus processos.

A contabilização do tempo de execução dos processos do 68020 é realizada pelo núcleo do 8088. Para que possa realizar esse controle, o 8088 possui em sua tabela de processos entradas correspondentes aos processos do 68020. Além disso, para saber qual o processo que atualmente está em execução no 68020, existe uma posição na memória do 68020, conhecida pelo 8088, com a identificação desse processo. A cada interrupção de relógio do 8088, a CLOCK-TASK verifica nessa posição qual o processo que está executando, somando nas variáveis de contabilização correspondentes a esse processo mais uma unidade de tempo. Se o mesmo processo estiver executando por mais de 100 ms, a CLOCK-TASK envia uma mensagem à SYS-TASK do 68020, informando que esse processo deve perder o processador. Ao receber essa mensagem, denominada `SYS_SCHED`, a SYS-TASK coloca o processo de usuário em execução no final da fila de processos prontos para executar, e entrega o processador para o próximo processo dessa fila.

### 3.3.5 Comunicação entre estações

Devido à inexistência de *hardware* dedicado à comunicação, foi necessário desenvolver um protocolo para a comunicação entre as estações, utilizando as interfaces paralelas existentes, originalmente destinadas à comunicação com impressoras. O protocolo originalmente projetado [BAR90] foi implementado e mostrou-se excessivamente lento (velocidade de pico: aproximadamente 6 kbytes por segundo) para ser utilizado na comunicação em um ambiente distribuído. Foi desenvolvido, então, um protocolo alternativo, mais rápido (pico de aproximadamente 285 kbytes por segundo). Esse protocolo está descrito no anexo A-1.

## 3.4 Proposta de Distribuição do DIX

Para transformar o MINIX em um sistema distribuído, o projeto DIX propôs o acréscimo de duas camadas ao sistema, ambas localizadas no processador 68020, logo abaixo da camada que contém os processos do usuário. Essa estrutura eatá representada na figura 3.4.

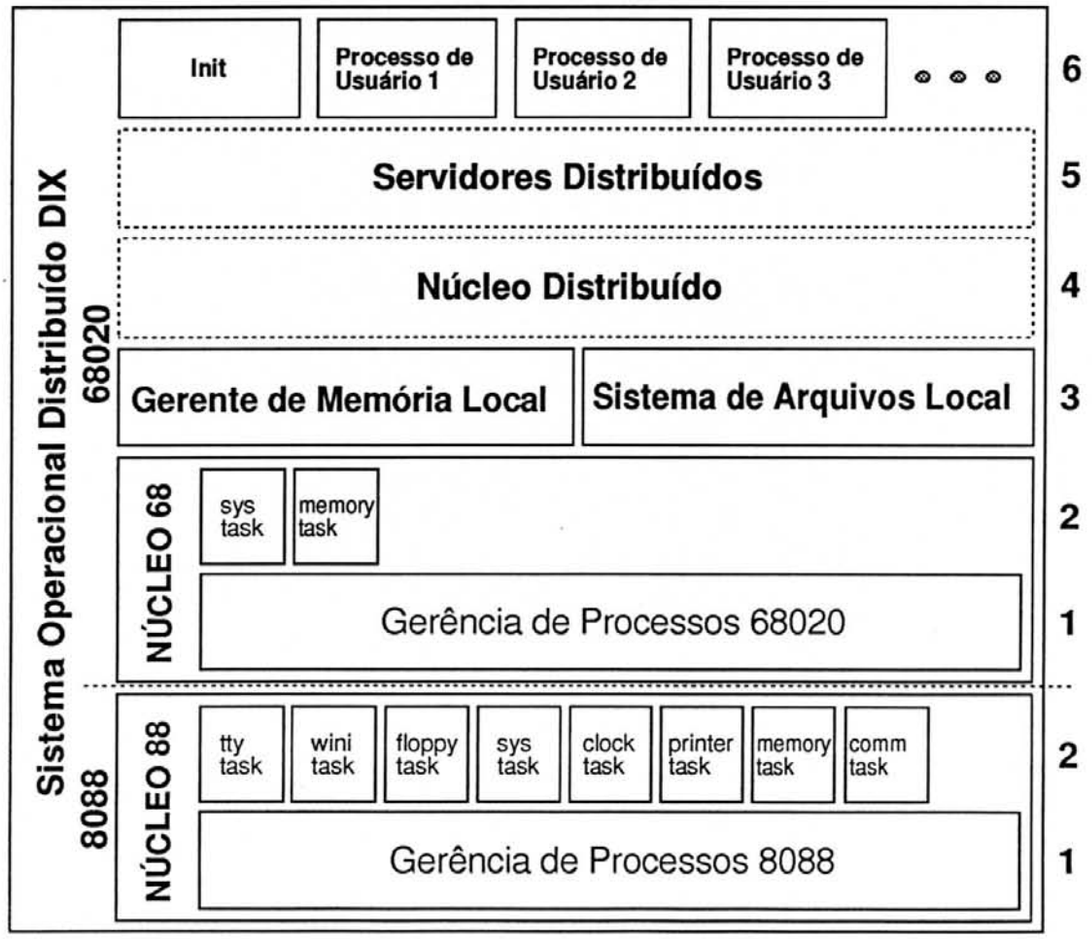


Figura 3.4: Estrutura do DIX

As três camadas inferiores correspondem às camadas originalmente presentes no MINIX, com a gerência de processos, as *tasks* e os servidores. Essas três camadas conhecem somente os recursos existentes na estação local e comunicam-se somente com processos locais.

A camada 4 é um novo núcleo, responsável por proporcionar comunicação transparente entre processos localizados em estações distintas. É a primeira camada a conhecer a existência de outras estações na rede. As mensagens das camadas superiores são desviadas para este núcleo, que verifica se a mensagem pode ser resolvida localmente ou se precisa ser enviada a outro nodo.

A camada 5 contém os servidores distribuídos. Esses servidores implementam as funções que provêm transparência de localidade. Esses servidores utilizam-se do núcleo distribuído da camada 4 e dos servidores locais das várias estações para atender os pedidos dos usuários. Inicialmente, existiriam dois servidores, o Executor Remoto de Processos (ERP) e o Servidor de Arquivos Global (SAG), em substituição ao MM e FS do MINIX, respectivamente. O ERP atende pedidos de criação, morte de processos, alocação de memória, etc. É responsável por decidir onde os processos serão criados e por manter a carga das estações razoavelmente equilibrada. O SAG realiza as operações de E/S requisitadas pelos processos de usuário. Deve implementar uma árvore única, igualmente visível por todas as estações, para o sistema de arquivos e diretórios.

Na camada superior, tal como no MINIX, estão os processos de usuário. Esses processos novamente não têm consciência da distribuição, que deve ser oferecida pelas camadas inferiores da forma mais transparente possível.

# 4 DEFINIÇÃO DO NÚCLEO DO SISTEMA

O núcleo de um sistema operacional deve ser mantido o mais simples possível [CHA90], a fim de que seja mais fácil depurá-lo, entendê-lo e portá-lo para outras plataformas. Dentro dessa filosofia, o núcleo proposto fornece somente os serviços indispensáveis para que servidores, externos a ele, possam tornar o sistema realmente distribuído, através da cooperação entre os nodos componentes da rede. Este capítulo descreve os serviços desse núcleo e a forma como o núcleo tolera determinadas falhas na rede de computadores.

## 4.1 Características Desejadas

Uma das principais características é que o núcleo seja pequeno, tendo como tarefa básica efetuar a comunicação entre os processos componentes do sistema. Para que o sistema possa ser distribuído, é importante que essa comunicação seja realizada de uma forma transparente à localização dos processos. Do ponto de vista dos processos que realizam a comunicação, o fato de os processos estarem localizados em um ou outro nodo deve ser irrelevante. A interface para comunicação com um processo remoto deve ser a mesma utilizada para a comunicação com processos locais. Na verdade, o processo sequer sabe se está se comunicando com um processo local ou remoto. O núcleo é responsável por localizar o processo destino e efetuar a troca de mensagens ou dados entre ele e o processo origem.

A criação e morte de processos no sistema é bastante dinâmica, resultando em uma distribuição não uniforme do processamento entre os nodos da rede. Alguns nodos serão sobrecarregados enquanto outros estarão sendo sub-utilizados. Uma forma de melhorar essa situação é criar os processos novos nos nodos menos carregados da rede. Dessa forma, a carga da rede tende a ser

equilibrada conforme são criados processos novos. Porém, a rede torna a se desequilibrar com a morte dos processos. Nada garante que os processos morrerão de forma a manter o equilíbrio. A forma de reequilibrar a rede neste caso é realizar a migração de processos dos nodos mais carregados aos que estiverem mais livres. Para isso, a transparência de localidade oferecida pelo núcleo do sistema deve suportar que os processos sejam migrados.

O núcleo provê somente os mecanismos que tornam possível a migração de processos. A política de migração, ou seja, a decisão sobre qual processo deve migrar, quando e para onde será implementada por um gerente de processos, externo ao núcleo. Esse gerente se utilizará de informações sobre a carga dos nodos da rede oferecidas pelo núcleo para tomar tais decisões de forma dinâmica. Uma outra possibilidade de uso da migração de processos visa a diminuição do tráfego da rede através da aproximação dos processos aos nodos que controlam os dispositivos de E/S que contêm os dados que tais processos utilizam.

O projeto do núcleo levou em conta uma outra característica importante: alterar o mínimo possível a interface original do sistema MINIX. O objetivo era manter o sistema em funcionamento sem alterar substancialmente as camadas superiores do sistema (os servidores e os utilitários). Dessa forma, nesta primeira fase somente o núcleo do sistema suporta a distribuição, enquanto os servidores continuam operando localmente, sem conhecimento da existência de outras estações na rede. A transformação do sistema em um sistema operacional realmente distribuído será efetuada de uma maneira gradual, conforme são desenvolvidos servidores que explorem as características distribuídas do núcleo.

Decidiu-se no início do projeto que não existiriam informações centralizadas. Nenhum dos nodos detém uma tabela com informações atualizadas sobre todos os processos atualmente em execução na rede. Caso existisse tal tabela, a tarefa de obter informações sobre um determinado processo seria extremamente facilitada. Bastaria uma simples consulta ao nodo que mantém a tabela. Por outro lado, manter uma tabela centralizada introduz ineficiências e diminui a

confiabilidade do sistema. Sempre que algum componente da rede altera ou consulta dados sobre um determinado processo, essa informação deve trafegar pela rede até o nodo onde a tabela centralizada se encontra. Isso acarreta atrasos e aumenta o tráfego da rede. Adicionalmente, se tal nodo falhar, o sistema como um todo pode se tornar inoperante. Com a tabela distribuída entre os nodos da rede, a tolerância a falhas é maior; quando algum nodo falha somente parte das informações é perdida, com o restante do sistema continuando operacional. É muito provável que as informações perdidas estejam replicadas em outros nodos, podendo ser recuperadas. A desvantagem reside na necessidade de um algoritmo mais complexo para localizar as informações desejadas no sistema.

O núcleo foi projetado para fornecer basicamente um mecanismo que permita que processos comuniquem-se de forma eficiente e confiável. Segundo Chandras [CHA90], a comunicação entre processos é talvez o componente mais importante de um sistema operacional distribuído. Essa comunicação fornece a base necessária para que processos servidores possam trocar informações entre si e prover os serviços que tornarão o sistema distribuído. Para efetuar a comunicação, é necessário uma forma de identificação de processos que os identifique globalmente no sistema. Tal identificação deve ser diferente da utilizada no MINIX, que não previa a existência de mais de um elemento processador no sistema. O identificador global de processos proposto está descrito na próxima seção.

## 4.2 Identificação de Processos

Para que um processo possa trocar informações, é necessário que o mesmo tenha como identificar precisamente o processo com o qual deseja se comunicar. Da mesma forma que no MINIX, os processos do sistema proposto são identificados por um número, denominado *pid*. A seguir, são descritas algumas

características desejáveis no identificador de processos, para ser utilizado em um sistema distribuído.

### 4.2.1 Características do identificador de processos

Os processos precisam ser identificados de forma única. Não deve ser possível que dois processos com o mesmo identificador estejam em execução ao mesmo tempo em toda a rede. Caso isso fosse permitido, o sistema e os demais processos não teriam como diferenciar tais processos, tornando impossível a comunicação.

Cada vez que um processo é criado no sistema, é gerado um identificador único para o mesmo. Para que a criação de processos possa ser realizada de forma eficiente, é necessário que a geração de identificadores também o seja. Uma forma de garantir a geração de um identificador único seria realizar uma consulta aos demais nodos da rede, para saber se determinado identificador já existe. Uma consulta desse tipo, além de ineficiente, provavelmente estaria sujeita a problemas causados pela geração concorrente de identificadores. Uma outra solução possível seria a geração centralizada de identificadores, o que vai de encontro à decisão de não centralizar informações no sistema. A melhor forma é a geração local dos identificadores, de forma independente, sem consulta a outros nodos da rede. No entanto, é necessário garantir que dois nodos distintos não gerarão o mesmo identificador para processos diferentes.

Conforme visto na seção 4.1, é desejável que processos possam migrar entre nodos. Quando isso acontecer, a identificação do processo que migra não deve mudar. Tal mudança acarretaria um tráfego muito grande de mensagens para informar a nova identificação a todos os processos com os quais o processo que migra mantém relações. Devido a essa característica, não é possível colocar junto à identificação do processo uma indicação do processador onde o mesmo se

encontra. Se isso fosse possível, a localização dos processos seria bastante simples. Apesar disso, é desejável que o processo seja facilmente localizado com base em sua identificação, para que a tarefa de roteamento de mensagens ou dados entre processos possa ser efetuada de forma eficiente.

Resumindo, as principais características desejáveis de um identificador de processos são:

- identificação unívoca do processo,
- eficiência na geração,
- imutabilidade,
- facilidade de localização do processo.

No MINIX, os processos são identificados por um número que corresponde simplesmente a sua entrada na tabela de processos, conhecida como *slot* [TAN87]. Esse método é bastante simples, apropriado para um sistema centralizado, mas não se adequa a um sistema distribuído, porque o mesmo *slot* existe em cada um dos nodos, e a identificação do processo não seria única. Devido a isso, não é possível utilizar a forma de identificação de processos original do MINIX.

### 4.2.2 Identificador de processos

O identificador de processos proposto é mostrado na figura 4.1. É dividido em duas partes, o número do nodo onde o processo foi criado e um número seqüencial. Esse identificador pode ser gerado localmente de forma eficiente, pois a identificação do nodo é conhecida, e a geração de um número seqüencial é trivial. Desde que os nodos tenham identificadores distintos, não existe a possibilidade de dois nodos gerarem identificadores de processos iguais.

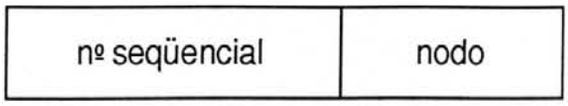


Figura 4.1: Identificador de processos

Essa forma de identificar processos é bastante semelhante à utilizada no sistema V, conforme visto na seção 2.2.1. A diferença está no significado do número do nodo. No sistema V esse número representa o local onde o processo está sendo executado, enquanto que no sistema proposto representa o local de criação do processo, que nunca muda. Para possibilitar a migração de processos no sistema proposto, foi desenvolvido um protocolo capaz de localizar um processo no sistema a partir de sua identificação. A próxima seção descreve esse protocolo.

## 4.3 Comunicação entre Processos

A principal tarefa do núcleo do sistema distribuído é realizar a comunicação entre os processos em execução no sistema. Para isso, o núcleo tem de localizar o nodo onde está sendo executado o processo destino. A localização é feita a partir da identificação do processo, devendo ser realizada de forma eficiente.

### 4.3.1 Localização dos processos

Para localizar os processos existe uma tabela, que contém o *pid* e o nodo onde se encontra o processo. O núcleo de cada nodo deve obrigatoriamente manter atualizadas as entradas de determinados processos nessa tabela. Outras entradas permanecem na tabela somente para acelerar a tarefa de localização de processos remotos, não havendo a necessidade de serem mantidas atualizadas. Os

processos cuja localização o núcleo tem a responsabilidade de manter atualizados nessa tabela são:

- processos em execução no nodo
- processos que foram criados no nodo

A necessidade de se conhecer a localização dos processos locais é óbvia—essa é a forma de o núcleo saber se um determinado processo encontra-se em execução em seu nodo ou não. Essa informação é facilmente mantida, pois o núcleo sempre é informado sobre a criação, morte ou migração de processos locais.

A razão para o núcleo conhecer a localização exata de todos os processos que foram criados no nodo é menos óbvia. Essa é a forma que o sistema tem de assegurar que ao menos um nodo sabe onde determinado processo se encontra. A identificação desse nodo é facilmente obtida, porque faz parte da identificação do processo. Para garantir que essa informação está sempre atualizada, toda vez que algum processo migra sua nova localização deve ser informada ao nodo onde o processo foi criado. Quando o processo morrer, esse fato também deverá ser comunicado ao nodo de criação do processo, que simplesmente eliminará a entrada correspondente ao processo de sua tabela. Como cada nodo obrigatoriamente conhece a localização de todos os processos criados localmente, um processo é considerado inexistente quando não for encontrado na tabela do nodo onde foi criado.

Além desses processos, a tabela contém a localização dos últimos processos que se comunicaram com algum processo local. Essa informação, não necessariamente atual, é utilizada pelo núcleo como uma sugestão para a localização de processos remotos. Na maioria dos casos, quando uma mensagem for enviada a um desses processos, este ainda estará no mesmo nodo em que residia na última vez que enviou uma mensagem.

Sempre que recebe uma mensagem para enviar a um processo, seja ela originada de um processo local ou de um outro nodo, o núcleo do sistema inicialmente verifica se o processo destino encontra-se na tabela. Se estiver presente, ele pode ser local ou remoto. Caso seja local, o envio da mensagem é processado da mesma forma que no MINIX original. Senão, a mensagem é enviada ao núcleo do nodo indicado pela tabela.

Caso o processo destino não seja encontrado na tabela, a mensagem é enviada ao nodo onde o processo foi criado, que obrigatoriamente sabe onde o processo se localiza. A identificação do nodo onde o processo foi criado não representa maiores dificuldades—é extraída diretamente de seu *pid*. Quando a mensagem finalmente chegar ao nodo onde o processo destino está sendo executado, a localização do processo origem é colocada na tabela desse nodo, de modo que a mensagem de resposta possa ser enviada diretamente ao nodo de onde a mensagem se originou. Quando o nodo origem receber a mensagem de resposta, colocará em sua tabela a localização do processo que a enviou. Caso os dois processos continuem a se comunicar, as próximas mensagens serão enviadas diretamente ao nodo correto, até que algum dos processos envolvidos seja migrado.

A figura 4.2 ilustra a comunicação entre dois processos, quando o processo destino não se encontra na tabela.

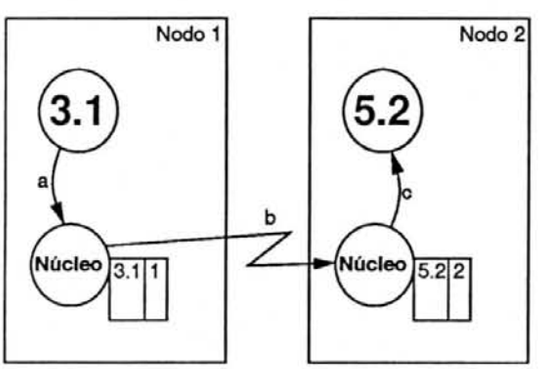


Figura 4.2: Comunicação entre dois processos

a. O processo 3.1 (onde 3 é o número seqüencial e 1 o nodo de criação) envia uma mensagem ao processo 5.2.

b. O núcleo do nodo 1 não encontra esse processo em sua tabela, e desvia a mensagem ao nodo 2, onde o processo foi criado.

c. Ao receber a mensagem, o núcleo do nodo 2 verifica que o processo 5.2 é local, e envia-lhe a mensagem.

O núcleo do nodo 2, ao receber a mensagem, atualiza sua tabela, inserindo uma entrada que registra que o processo 3.1 está localizado no nodo 1. Quando o processo 5.2 responder ao processo 3.1, o nodo destino será localizado diretamente.

### 4.3.2 Migração de processos

Quando um processo migra, três nodos são envolvidos: o nodo onde o processo está, o nodo para onde o processo migra e o nodo onde o processo foi criado. Esses três nodos não são necessariamente distintos—o processo pode estar migrando de ou para o nodo onde foi criado. Antes que a migração seja concretizada, o nodo onde o processo foi criado é informado da nova localização do processo, para que possa atualizar sua tabela. O nodo que executava o processo também atualiza sua tabela.

Os nodos não envolvidos com a migração não são avisados. Os nodos que não contêm processos em comunicação com o processo que migrou não serão afetados. As tabelas dos demais nodos serão atualizadas conforme haja comunicação com o processo que migrou. Se o processo que migrou enviar uma mensagem a algum processo local, a tabela será atualizada, sem maiores problemas. Se antes disso um processo local enviar uma mensagem ao processo que migrou, o núcleo enviará essa mensagem ao nodo onde o processo estava antes de

migrar, conforme informa sua tabela. Provavelmente esse nodo ainda tenha em sua tabela o local para onde o processo migrou, caso em que desviará a mensagem para lá. Se tal informação não estiver mais em sua tabela, a mensagem será desviada para o nodo onde o processo que migrou foi criado, que obrigatoriamente tem a localização atual do processo. Quando o processo que migrou responder, a tabela será atualizada, e as próximas mensagens entre os dois processos serão enviadas diretamente aos nodos corretos.

Somente uma mensagem (por nodo envolvido) passa por nodos intermediários da rede a cada vez que há migração de um processo, sendo a comunicação direta entre os dois nodos envolvidos rápida e facilmente reestabelecida. A seguir, um exemplo de comunicação entre processos quando um dos processos migra.

A figura 4.3 mostra o envio de uma mensagem do processo 3.1, no nodo 1, ao processo 5.2, que migrou do nodo 2 ao nodo 3. Na tabela do nodo 1, o processo 5.2 consta como estando no nodo 2.

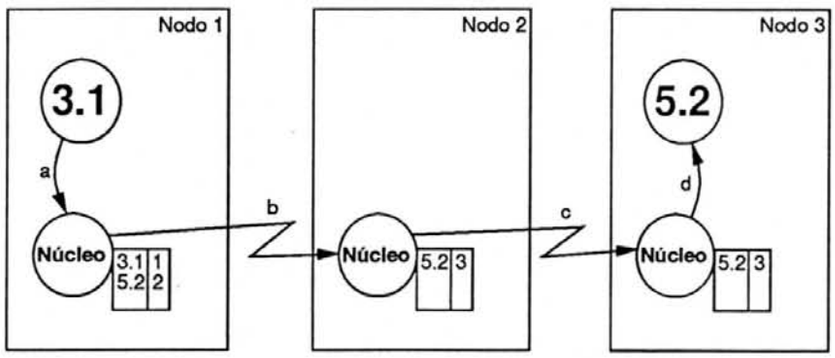


Figura 4.3: Comunicação entre dois processos, após migração

a. a mensagem é enviada pelo processo 3.1 ao processo 5.2

b. o núcleo do nodo 1 envia a mensagem para o nodo 2, conforme informação contida em sua tabela

c. ao receber a mensagem, o núcleo do nodo 2 analisa sua tabela e verifica que o processo 5.2 encontra-se no nodo 3 (o nodo 2 obrigatoriamente

tem essa informação, pois é o local de criação do processo 5.2), e a mensagem é mais uma vez desviada

d. ao chegar no nodo 3, a mensagem é enviada ao processo destino

O nodo 3 atualiza sua tabela, tão logo recebe a mensagem, para conter a localização do processo 3.1. Quando o processo 5.2 responder, sua mensagem será enviada diretamente ao nodo 1, que então conhecerá a nova localização do processo 5.2. Enquanto não houver nova migração, a comunicação entre os dois processos será direta.

Suponhamos que, devido ao aumento de carga no nodo 3, o processo 5.2 seja novamente migrado, desta vez para o nodo 4. O nodo 2, por ser o local de criação do processo, é informado da migração. Suponhamos também que o tráfego de mensagens no nodo 3 seja grande, e que sua tabela não contenha mais a localização do processo 5.2. Essa situação está esquematizada na figura 4.4.

a. o processo 3.1, no nodo 1, envia uma mensagem ao processo 5.2

b. a mensagem é desviada para o nodo 3, que é a localização do processo 5.2 informada pela tabela do nodo 1

c. o nodo 3 não encontra o processo 5.2 em sua tabela, e desvia a mensagem para o nodo 2, local de criação do processo, que conhece a localização do processo

d. o nodo 2 envia a mensagem para o nodo 4, localização atual do processo destino

e. o núcleo do nodo 4 finalmente entrega a mensagem ao processo 5.2.

Da mesma forma que nos casos anteriores, a tabela do nodo 4 será atualizada e a resposta será enviada diretamente ao nodo 1, que por sua vez atualizará sua própria tabela.

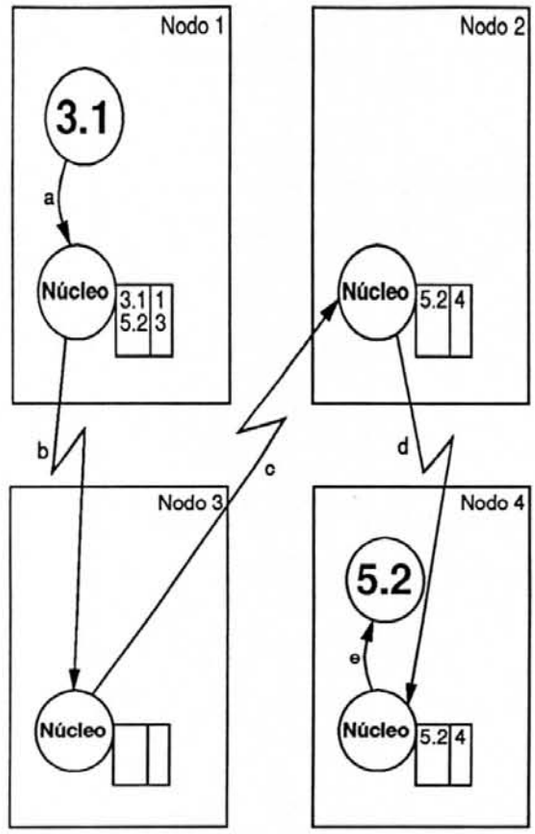


Figura 4.4: Comunicação entre dois processos, após segunda migração

À primeira vista, pode parecer que uma mensagem poderia trafegar indefinidamente entre vários nodos sem nunca chegar a seu destino. Para que um ciclo como esse acontecesse, todos os nodos por onde a mensagem está trafegando deveriam conter em sua tabela uma informação (falsa) sobre a localização do processo destino da mensagem. Se um dos nodos não possuísse tal informação, desviaria a mensagem ao nodo onde o processo foi criado, que conhece obrigatoriamente a localização atual do processo. Além disso, o processo não poderia estar em nenhum dos nodos componentes do ciclo, ou a mensagem seria entregue ao processo quando chegasse a esse nodo (os nodos mantêm obrigatoriamente em sua tabela a localização dos processos locais).

Sempre que uma informação sobre a localização de um processo é colocada na tabela, ela é verdadeira. Logo, a informação contida em todos os nodos que constituem o ciclo deve ter sido verdadeira em algum momento, ou seja, o processo destino necessariamente esteve em cada um desses nodos, ou o

ciclo não existiria. Quando o processo deixou o último dos nodos constituintes do ciclo, esse nodo obrigatoriamente soube o local para onde o processo migrou. Se esse nodo receber a mensagem enquanto ainda detém essa informação, a mensagem será desviada ao nodo para onde o processo migrou. Caso contrário, o nodo não tem nenhuma informação, e enviará a mensagem ao nodo de criação do processo. De qualquer forma, a mensagem chegará ao processo destino, não sendo possível que o ciclo seja formado.

## 4.4 Falha em um Nodo

Um dos aspectos mais atraentes de um sistema operacional distribuído é a possibilidade de aproveitar-se da replicação de recursos para prover tolerância a falhas. Sendo um componente fundamental, o núcleo de um sistema operacional distribuído também deve ser tolerante a falhas. As falhas endereçadas pelo sistema proposto são falhas totais de um nodo—caso em que o nodo não é mais acessável e todos os processos que estavam sendo executados nesse nodo são perdidos. Não é responsibilidade do núcleo do sistema recuperar os processos que estavam sendo executados no nodo que falhou. Isso seria tarefa de um gerente de processos tolerante a falhas externo ao núcleo, como o proposto por Belmonte [BEL92]. É responsabilidade do núcleo, no entanto, garantir a continuidade de comunicação entre os processos não diretamente envolvidos com o nodo que falhou.

A principal informação perdida quando um nodo falha, do ponto de vista do núcleo distribuído, é a localização corrente dos processos que foram criados nesse nodo. A responsabilidade pela recuperação e manutenção de tal informação é herdada por outro nodo. A identificação desse nodo herdeiro deve ser fácil de se obter a partir da identificação do processo, para evitar que todos os nodos da rede tenham que ser informados e que tenham que manter essa informação em suas tabelas. O nodo escolhido é o que tem a maior identificação

menor que a do nodo que falhou (à exceção do nodo com a menor identificação na rede, que é herdado pelo que possui a maior). Assim, por exemplo, supondo que existam três nodos, chamados 1, 2 e 3, o nodo 1 é herdeiro do 2, que é herdeiro do 3, que por sua vez é o herdeiro do nodo 1. Essa escolha foi tomada porque, no protótipo implementado, esse é o nodo que envia o *token* da rede ao nodo falho (ver anexo A-1), e provavelmente será o primeiro nodo a perceber a falha. Além disso, é uma forma bastante simples de escolha.

Quando um nodo N não consegue enviar uma mensagem para um nodo C onde determinado processo foi criado, repassa essa mensagem para o nodo H, herdeiro de C. Se o nodo herdeiro não souber a localização do processo, tenta enviar a mensagem ao nodo C onde o processo foi criado, que deve saber. Ao perceber o nodo C não está mais disponível, o nodo H torna-se herdeiro, e deve localizar o processo destino, bem como os demais processos cuja localização era responsabilidade do nodo C, que falhou. Para isso, o nodo H envia uma mensagem para todos os demais nodos, informando a falha e perguntando sobre o paradeiro dos processos. Os nodos que estiverem executando algum processo criado no nodo C (ou por ele herdado), enviam uma mensagem ao nodo H informando esse fato. As entradas nas tabelas que informavam o nodo C como local de execução corrente de algum processo são removidas, mesmo pelos nodos onde tais processos foram criados. Dessa forma, esses processos passam a ser considerados mortos. Ao final, todos os nodos da rede tomarão conhecimento da falha e automaticamente desviarão as mensagens destinadas ao nodo falho C para o nodo herdeiro H.

No caso de existir um gerente de processos distribuídos, o núcleo herdeiro esperará para ser informado pelo gerente da localização dos processos herdados, inclusive os eventualmente recuperados, no caso de o gerente prover tolerância a falhas.

Além dos processos em execução no nodo e dos processos nele criados, cada nodo também mantém, obrigatoriamente, a localização dos processos her-

dados de nodos falhos. O nodo herdeiro deve ser informado sobre migrações e mortes dos processos que herdou.

Para exemplificar a recuperação do sistema em caso de falha, suponhamos que o processo 6.3, criado e ainda localizado no nodo 3 envia uma mensagem ao processo 4.2, que foi criado no nodo 2 e migrou para o nodo 4. Houve uma falha no nodo 2, e o único nodo da rede que sabe a localização do processo 4.2 é o nodo 4, onde o processo se encontra em execução. Além disso, a falha do nodo 2 ainda é desconhecida pelos demais nodos da rede. A figura 4.5 ilustra essa configuração.

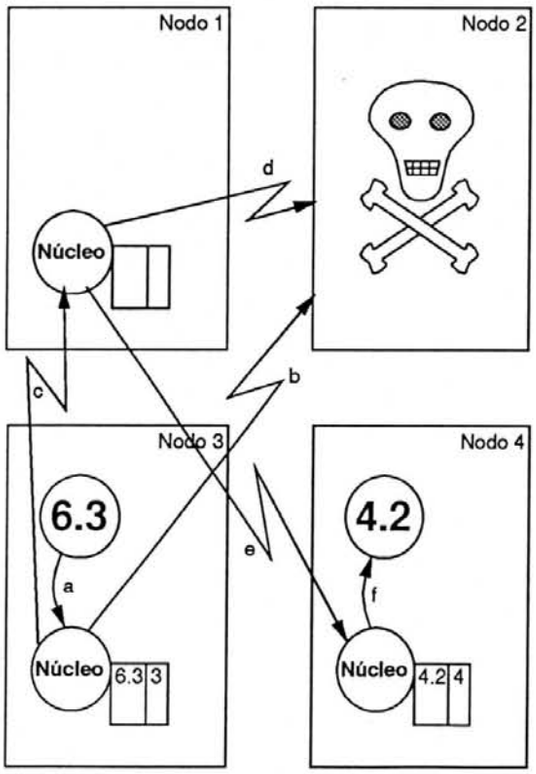


Figura 4.5: Comunicação entre dois processos, após falha em um nodo

a. o processo 6.3, no nodo 3, envia uma mensagem ao processo 4.2

b. o núcleo do nodo 3, por não saber a localização do processo 4.2, envia a mensagem ao nodo 2, onde o processo foi criado. A comunicação não tem êxito, pois o nodo 2 falhou.

c. a mensagem é então enviada ao nodo 1, herdeiro do nodo 2.

d. o nodo 1, por desconhecer a localização do processo 4.2, também tenta enviar a mensagem ao nodo 2, quando descobre a falha.

Nesse momento, o nodo 1 percebe que é o herdeiro do nodo falho e comunica-se com os demais nodos da rede, para descobrir a localização atual dos processos criados (ou herdados) no nodo 2. Com isso, todos os nodos da rede perceberão a falha do nodo 2. O nodo 4 responde ao nodo 1, informando que está executando o processo 4.2, criado no nodo falho. O nodo 1 então envia a mensagem ao nodo 4, e a operação da rede é normalizada. Caso nenhum nodo respondesse afirmando que contém o processo 4.2, ele seria considerado morto e um erro seria devolvido ao processo 6.3.

Um esquema semelhante de herança também é utilizado quando nodos são desligados da rede de forma programada. A diferença nesse caso é que o nodo herdeiro é informado da herança e da localização dos processos que herda. Os processos em execução no nodo que é desligado são previamente migrados para outros nodos. A informação da herança, bem como a migração dos processos, seria controlada por um servidor de processos distribuído. Pode também haver migração de dados contidos em periféricos que serão desconectados da rede por estarem sendo controlados pelo nodo que será desligado. Essa migração de dados seria tarefa de um servidor de arquivos distribuído.

## 4.5 Entrada de um Nodo na Rede

Eventualmente, um nodo tem de ser conectado a uma rede ativa. Isso acontece normalmente durante a inicialização do sistema, mas pode acontecer em qualquer outro momento. O protocolo de comunicação deve ser capaz de reorganizar-se rapidamente para refletir a mudança na configuração da rede.

Após essa reorganização, o novo nodo é visto como se estivesse todo o tempo conectado à rede. Inicialmente, o novo nodo informa estar disposto a entrar na rede a seu nodo herdeiro. Os processos que o novo nodo teria herdado caso estivesse todo o tempo conectado à rede foram herdados por seu nodo herdeiro. O nodo herdeiro passa então ao novo nodo a responsabilidade de localizar os processos que o novo nodo teria herdado.

Por exemplo, suponhamos que somente os nodos 2, 5 e 12 estão ativos em uma rede à qual já estiveram também conectados os nodos 8 e 10. Nesse caso, o nodo 5 herdou os processos dos nodos 8 e 10. Caso o nodo 9 seja conectado à rede nesse momento, o nodo 5 deve lhe passar as informações que herdou do nodo 10, que teriam sido transferidas ao nodo 9, caso estivesse conectado à rede quando o nodo 10 foi desligado. Não deve, entretanto, repassar as informações relativas ao nodo 8, porque, mesmo que o nodo 9 estivesse ativo no momento em que o nodo 8 foi desligado, o nodo herdeiro ainda seria o nodo 5.

Os gerentes distribuídos também são informados da conexão do novo nodo, para que possam utilizar-se dos recursos oferecidos por ele. Conforme os processos criados (ou migrados) no novo nodo comuniquem-se com outros processos do sistema, os demais nodos saberão de sua existência, e passarão a considerá-lo nos procedimentos de localização de processos, normalizando o esquema de comunicação da rede.

# 5 IMPLEMENTAÇÃO DO PROTÓTIPO DO SISTEMA

Este capítulo descreve as *tasks* que foram adicionadas ao sistema MINIX para implementar os algoritmos descritos no capítulo 4, transformando o núcleo do MINIX em um núcleo distribuído. O protótipo do sistema foi implementado tendo como base o trabalho desenvolvido durante o projeto DIX (ver capítulo 3).

A estrutura proposta no projeto DIX para tornar o MINIX distribuído previa a inclusão de duas novas camadas ao sistema, entre os servidores locais e os processos de usuários: um núcleo distribuído e os servidores globais. O núcleo distribuído seria localizado acima dos processos servidores locais, e seria responsável por tornar transparente a existência de uma rede de computadores. O tráfego de informações entre esse núcleo distribuído e o núcleo original necessariamente seria grande, rompendo a estrutura modular do MINIX, na qual um processo pode se comunicar somente com processos localizados na camada imediatamente inferior à sua. No protótipo implementado, esse núcleo, ao invés de ser mais uma camada do sistema, foi incorporado ao núcleo local, por meio da inclusão de três novas *tasks* ao sistema:

MSG-TASK para o roteamento de mensagens entre processos,

XFER-TASK para transferência de blocos de dados entre processos e

COMM-TASK para realizar a comunicação entre os nodos da rede.

A MSG-TASK implementa o algoritmo de localização de processos descrito no capítulo 4. As mensagens enviadas por processos locais, ou recebidas de outros nodos pela COMM-TASK são desviadas para essa *task*. Em linhas gerais, a MSG-TASK verifica se o destino da mensagem é local ou remoto, e entrega a mensagem ao processo, ou ao nodo onde supostamente o processo está localizado.

Além disso deve oferecer uma certa garantia de que as mensagens chegarão a seu destino, mesmo que possíveis falhas no subsistema de comunicação façam com que algumas mensagens sejam perdidas.

A XFER-TASK é usada para as cópias de dados entre processos. Essas cópias são geralmente realizadas a pedido dos servidores, para atender solicitações dos processos de usuário, tais como funções de E/S. Os servidores, nesse caso, não sabem se o processo que estão servindo é local ou remoto. A função da XFER-TASK é basicamente substituir a chamada SYS_COPY realizada pela SYS-TASK, utilizada no MINIX para cópias desse tipo—ver seção 3.3.2. Essa chamada continua existindo, mas realiza cópias somente entre processos locais. Ao receber uma mensagem para cópia de dados, a XFER-TASK verifica se a cópia é local ou não. Se a cópia for local, o pedido é repassado à SYS-TASK. Caso contrário, a cópia é realizada com a ajuda da COMM-TASK.

As cópias de dados ou mensagens entre os nodos são realizadas pela COMM-TASK. Sua tarefa é isolar das demais *tasks* os detalhes da comunicação com os outros nodos, oferecendo uma interface simples para a realização dessa comunicação. Esta *task* controla o *hardware* da interface de comunicação, conforme descrito no anexo A-1.

A estrutura do sistema, após a inclusão dessas novas *tasks* é mostrada na figura 5.1. As camadas 1, 2 e 3 são idênticas às camadas correspondentes do MINIX, à exceção das novas *tasks*. A camada 4 contém os servidores distribuídos. Esses servidores atendem os pedidos dos processos de usuário, agora localizados na camada 5, utilizando-se dos servidores locais da camada 3 e dos servidores distribuídos localizados nos demais nodos.

Não foram implementados mecanismos para a migração de processos. Devido à inexistência de uma Unidade de Gerência de Memória nas estações Proceda, os processos em execução no 68020 realizam acessos a endereços absolutos da memória do 68020. Um processo não pode ser executado em um endereço

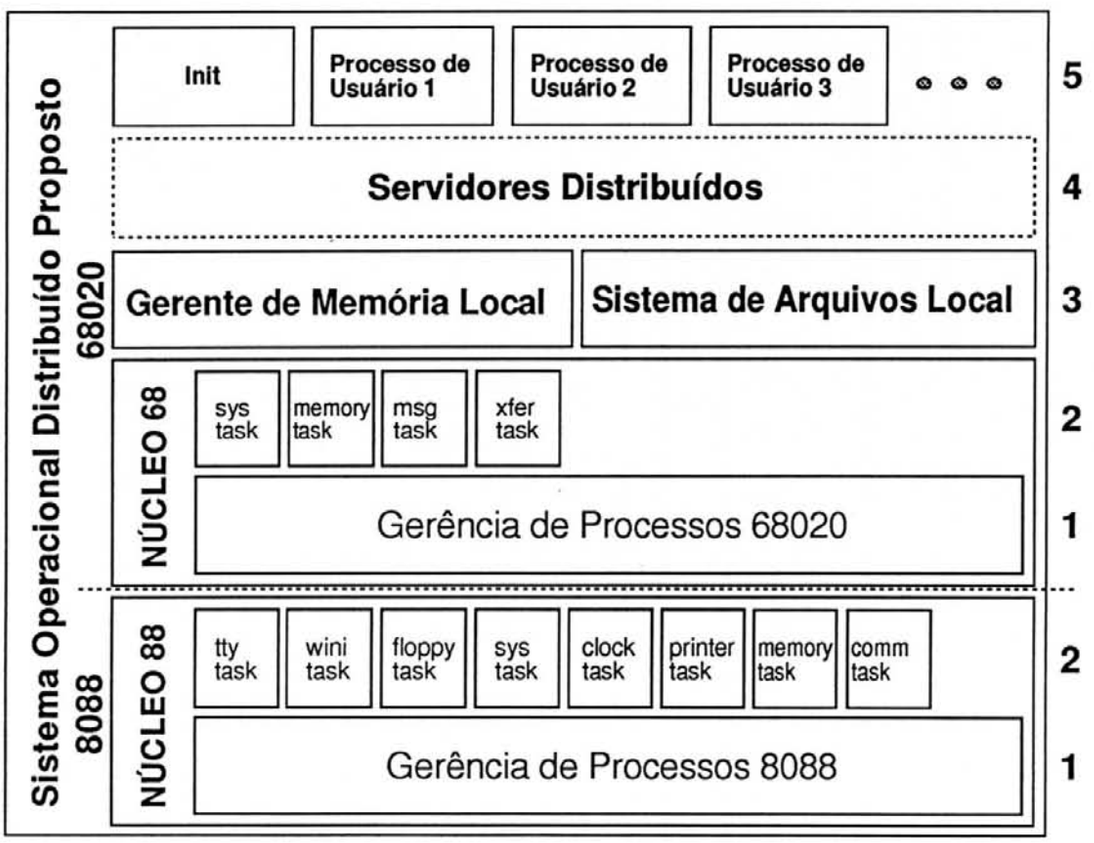


Figura 5.1: Estrutura do sistema proposto

diferente do que foi inicialmente carregado, quando foi realizada uma relocação estática de todas suas referências. Para que pudesse migrar, seria necessário que o processo fosse colocado no processador destino no mesmo endereço físico que ocupava no processador destino. Essa restrição torna impraticável a migração de processos no *hardware* das estações Proceda. O sistema de troca de mensagens, no entanto, foi implementado com todo o suporte à migração.

As três *tasks* que foram incluídas no sistema são descritas mais detalhadamente nas próximas seções.

## 5.1 A *Task* de Mensagens

Na *task* de mensagens, denominada MSG-TASK, está implementado o algoritmo de localização de processos descrito no capítulo 4. O conjunto de MSG-TASKs do sistema é responsável por saber a localização exata de cada processo em execução no sistema e de efetuar as trocas de mensagens entre esses processos. Todas as mensagens geradas por processos locais ou desviadas para o nodo local por alguma MSG-TASK remota são processadas por esta *task*, que decide se a mensagem deve ser enviada a um processo local ou a outra MSG-TASK. Esta *task* está localizada no processador 68020, onde estão os processos de usuário e os servidores, processos responsáveis pela maioria do tráfego de mensagens não locais no sistema.

A MSG-TASK mantém a tabela de localização de processos descrita na seção 4.3. As mensagens provenientes de um processo local são interceptadas pelo núcleo do sistema e enviadas à MSG-TASK. Mensagens recebidas pela COMM-TASK provindas de outros nodos são também enviadas à MSG-TASK. A MSG-TASK analisa essas mensagens e verifica a localização do processo destino em sua tabela. Se o processo for local, a mensagem segue o rumo normal que seguiria no MINIX, ou seja, o sistema verifica se o processo destino está esperando uma mensagem

do processo origem e, caso esteja, a mensagem lhe é enviada. Se o processo não estiver esperando por uma mensagem, ou se estiver esperando uma mensagem de um outro processo, a mensagem é armazenada em um *buffer* interno ao núcleo. Essa mensagem será entregue ao processo destino tão logo o mesmo efetue um `receive` para receber uma mensagem do processo origem (ou do identificador especial ANY, que significa qualquer processo).

Se a tabela indicar que o processo destino encontra-se em outro nodo, a MSG-TASK faz um pedido à COMM-TASK para que esta envie a mensagem ao nodo indicado. A COMM-TASK do nodo destino enviará a mensagem à MSG-TASK de seu nodo, que procederá de forma análoga à MSG-TASK origem.

No caso de o processo destino não constar na tabela, a MSG-TASK extrai o nodo de criação do processo de seu *pid*, e envia a mensagem (através da COMM-TASK) a esse nodo. Se o processo foi criado no nodo local, é considerado morto—se estivesse vivo, obrigatoriamente estaria na tabela. Nesse caso, uma mensagem de erro é enviada ao nodo do processo que enviou a mensagem.

A MSG-TASK pode receber mensagens de 8 tipos diferentes:

`NODE_ID` enviada pela COMM-TASK durante a inicialização, para informar a identificação do nodo local

`XMSG_RCVD` enviada pela COMM-TASK quando esta receber uma mensagem de outro nodo

`LMSG_RCVD` enviada pelo núcleo local quando um processo local enviar uma mensagem a outro processo

`XMSG_ACK` informa o correto recebimento, por um nodo remoto, de uma mensagem originada neste nodo

`XMSG_ERROR` enviada por outra MSG-TASK quando houver erro na entrega de uma mensagem remota

PROC_LCTN usada para informar a localização de um processo que este nodo tem a responsabilidade de saber

DEAD_NODE enviada pela COMM-TASK local quando não conseguir passar o *token* para o próximo nodo; utilizada também para informar as demais MSG-TASKs a morte de um nodo, pedindo informações sobre processos herdados

INH_PROC enviada por MSG-TASKs remotas para informar a localização de processos herdados

Antes de poder operar normalmente, a MSG-TASK aguarda uma mensagem do tipo NODE_ID, que contém a identificação do nodo local. Sem essa identificação, a MSG-TASK não tem como saber se um processo é local ou remoto. Essa mensagem é enviada pela COMM-TASK durante sua inicialização.

A mensagem do tipo XMSG_RCVD, recebida da COMM-TASK, informa a recepção de uma mensagem remota, contida em um pacote de dados. A mensagem XMSG_RCVD é composta pelos seguintes campos:

NODE contém o nodo de onde o pacote de dados vem (último nodo por onde o pacote passou, no caso de ser uma mensagem que passou por vários nodos)

ADDRESS contém o endereço, dentro do espaço de endereçamento da COMM-TASK, onde o pacote de dados está armazenado

COUNT contém o tamanho do pacote de dados, em bytes

Ao receber uma mensagem desse tipo, a MSG-TASK primeiramente efetua uma cópia do pacote de dados apontado pelo campo ADDRESS para um *buffer* em seu próprio espaço de endereçamento e envia uma mensagem de resposta à COMM-TASK, que pode então liberar seu *buffer* e prosseguir normalmente. Essa

cópia é necessária porque as duas *tasks* estão localizadas em processadores diferentes, sem acesso direto à memória uma da outra.

O pacote de dados contém, além da mensagem propriamente dita, o nodo onde está localizado o processo que enviou a mensagem e um número que identifica a mensagem (utilizado para confirmação de recepção ou para reconhecer a mensagem no caso de ser uma retransmissão—ver adiante). A informação sobre o nodo do processo é utilizada pela MSG-TASK para atualizar sua tabela de localização de processos. Se o processo destino não for local, o pacote com a mensagem é novamente repassado à COMM-TASK, para que o envie ao nodo indicado na tabela (ou ao nodo onde o processo destino foi criado, caso não conste na tabela).

Se o processo for local, a mensagem é extraída do pacote recebido da COMM-TASK e enviada ao processo (ou armazenada, caso o processo não esteja pronto para receber). Então, a tabela de localização de processos é atualizada. Caso o processo que enviou a mensagem já conste na tabela, sua localização é simplesmente sobreposta à informação presente na tabela. Se não existir uma entrada referente ao processos na tabela, tal entrada deve ser criada. Se a tabela estiver cheia, algum outro processo deve ser retirado da tabela para criar espaço. O processo escolhido é o que está a mais tempo na tabela sem ser referenciado, entre os processos cuja localização o nodo não tem a responsabilidade de manter atualizada—ver seção 4.4.

A mensagem do tipo `LMSG_RCVD`, recebida do núcleo local, contém somente o campo `ADDRESS`, que informa o endereço onde está armazenada a mensagem. Como esse endereço pertence ao espaço de endereçamento do núcleo local, localizado no processador 68020, a mensagem pode ser acessada diretamente pela MSG-TASK, sem a necessidade de ser copiada para um *buffer* local. Essa mensagem é processada de forma bastante semelhante à `XMSG_RCVD` descrita anteriormente. A diferença está na localização do processo origem, que é conhecida (é o próprio nodo local), não havendo, portanto, necessidade de atualizar a tabela

de localização de processos. Caso a mensagem tenha de ser enviada a um outro nodo, é colocada em um pacote de dados, juntamente com um número seqüencial que a identifica e a identificação nodo do processo origem da mensagem, ou seja, o nodo local.

Para manter a compatibilidade com o MINIX, o envio de mensagens é síncrono, ou seja, um processo é bloqueado quando envia uma mensagem, sendo liberado somente quando sua mensagem é recebida pelo processo destino. Enquanto a recepção de uma mensagem remota não é acusada pela MSG-TASK do nodo destino, essa mensagem é mantida na tabela de processos do núcleo, em uma posição correspondente ao processo que envia a mensagem. O recebimento da mensagem pode ser acusado na forma de uma resposta do processo destino, ou através de uma mensagem do tipo XMSG_ACK, enviada pela MSG-TASK do nodo destino quando o processo destino recebeu a mensagem mas demora a responder. Se o recebimento não for acusado dentro de um determinado período de tempo, a MSG-TASK origem retransmite a mensagem. Como as mensagens têm identificação, é possível para o nodo destino identificar quando uma mensagem vinda de outro nodo trata-se da retransmissão de uma mensagem já recebida. Nesse caso, a MSG-TASK destino simplesmente ignora a retransmissão, e envia uma mensagem do tipo XMSG_ACK para confirmar a recepção da mensagem.

A mensagem do tipo XMSG_ERROR é recebida quando alguma mensagem originada neste nodo não pode ser enviada ao processo destino, provavelmente porque esse processo não existe mais. A mensagem contém a identificação do processo cuja mensagem não pode ser enviada, e este processo é desbloqueado com um código de erro (o processo está bloqueado, porque sua mensagem ainda não foi confirmada).

Quando um processo é criado, migra ou é terminado, o nodo responsável por saber a localização desse processo é informado, para que possa manter atualizada sua tabela de localização de processos. Para tanto, é utilizada a mensagem PROC_LCTN, que contém a identificação do processo, sua localização

e um código que informa se o processo foi criado, migrou ou morreu. Ao receber essa mensagem, a MSG-TASK atualiza sua tabela com a localização do processo. A forma de o nodo responsável pela localização de um processo saber se esse processo está morto é simplesmente ele não estar presente na tabela. Assim, ao receber uma mensagem desse tipo indicando que um determinado processo morreu, a MSG-TASK simplesmente exclui da tabela a entrada correspondente ao processo que morreu.

Quando a COMM-TASK não conseguir, após várias tentativas, enviar o *token* ao próximo nodo da rede (ver seção 5.3 e anexo A-1), assume que esse nodo está morto, e envia uma mensagem do tipo `DEAD_NODE` à MSG-TASK. Ao receber essa mensagem, a MSG-TASK deverá herdar do nodo morto as informações referentes aos processos cuja localização era responsabilidade desse nodo. Para isso, envia uma mensagem do tipo `DEAD_NODE` às MSG-TASKs dos demais nodos da rede. Ao receber uma mensagem do tipo `DEAD_NODE` de outra MSG-TASK, a MSG-TASK local verifica se está sendo executado em seu nodo algum processo que nasceu ou que foi herdado pelo nodo morto, e envia essa informação, em mensagens do tipo `INH_PROC`, à MSG-TASK que fez o pedido. Além disso, sempre que receber uma mensagem desse tipo, retira de sua tabela todos os processos que estão marcados como executando no nodo que morreu. Retira também o nodo que acaba de morrer de sua lista de nodos vivos. Ao final, todos os nodos terão conhecimento da morte, e o nodo herdeiro terá herdado todos os processos que estavam sob responsabilidade do nodo que falhou.

## 5.2 A *Task* de Cópia de Dados

Além da troca de mensagens, os processos podem realizar trocas de quantidades maiores de dados. Isso pode ocorrer, por exemplo, quando um processo lê um bloco de dados de um arquivo em disco. Um pedido de leitura como esse é tratado pelo servidor de arquivos, que deve contar com mecanismos

para armazenar os dados lidos do disco no espaço de endereçamento do processo que fez o pedido. Para realizar cópias desse tipo, foi criada a XFER-TASK. No MINIX, tais cópias são efetuadas pela SYS-TASK, através da chamada `SYS_COPY`. Essa chamada continua existindo, mas é capaz de realizar somente cópias entre processos locais. A função da XFER-TASK é receber pedidos de copias de dados, analisar a localização do processo destino e, caso o processo seja local, repassar o pedido à SYS-TASK. Se o destino dos dados for remoto, a XFER-TASK efetua a transferência de dados utilizando-se dos serviços da COMM-TASK.

Para evitar que um volume grande de dados tenha que ser roteado no caso de um processo que fez um pedido de dados migrar antes de o pedido ser atendido, não é permitido que um processo migre quando tem um pedido pendente em algum servidor do sistema. Para simplificar ainda mais a situação, um processo só pode migrar quando estiver executando ou na fila de processos prontos para execução.

Conforme será visto adiante, a COMM-TASK aceita somente pedidos de cópia de dados que tenham como origem o nodo local e como destino um outro nodo, não sendo possível ler dados de um nodo remoto com um pedido direto a essa *task*. Para resolver esse problema, a XFER-TASK transforma os pedidos de leitura de dados remotos em pedidos à XFER-TASK do nodo remoto que contém os dados. Será a XFER-TASK remota que efetuará o pedido de cópia à COMM-TASK. Esse pedido será, então, um pedido de envio de dados, aceito pela COMM-TASK.

## 5.3 A *Task* de Comunicação entre Nodos

A comunicação entre os nodos, seja de mensagens ou de blocos de dados, é realizada, em última instância, pela COMM-TASK. Essa *task* recebe pedidos de cópia tanto da MSG-TASK quanto da XFER-TASK, sendo responsável por efetuar uma multiplexação da via física de comunicação para efetivar esses pedidos. A

COMM-TASK está dividida em dois níveis: o nível superior, que recebe os pedidos de cópia e os divide em pequenos pacotes, e o inferior, que controla a interface de comunicação e envia ou recebe esses pacotes. É no nível inferior que estão os mecanismos para envio e recebimento do *token*, que controla qual o nodo que pode transmitir dados a cada momento. O nível inferior da *task* está descrito no anexo A-1.

O nível superior da COMM-TASK aceita três tipos de mensagens:

`SEND_DATA` enviada pela XFER-TASK quando esta deseja transmitir um bloco de dados a outro nodo,

`SEND_MSG` pedido de envio de mensagem remota feito pela MSG-TASK,

`COMM_INT` enviada pelo nível inferior da *task* quando algum dado ou o *token* foi recebido de um nodo remoto

Os pedidos de transmissão, tanto de dados quanto de mensagens, recebidos pela COMM-TASK são transformados em pequenos pacotes, que o nível inferior da *task* é capaz de transmitir. Esses pacotes são enfileirados por ordem de chegada, e serão transmitidos tão logo a *task* esteja de posse do *token*, o que lhe garante o direito de transmissão.

A chegada do *token*, bem como a chegada de pacotes de dados provenientes de outros nodos são informados pelo nível inferior da *task* através de mensagens do tipo `COMM_INT`. No caso de ser informado que está de posse do *token*, o nível superior da *task* passa a enviar, em ordem, os pacotes de dados que estão armazenados aguardando envio. Cada nodo pode transmitir um número limitado de pacotes de dados a cada vez que recebe o *token*. Caso não houvesse tal restrição, um nodo poderia ficar de posse do *token* por um tempo excessivo, prejudicando a comunicação dos demais nodos. Tão logo transmita esse número de pacotes de dados, ou quando não possuir mais pacotes a transmitir, a COMM-TASK passa o *token* para o próximo nodo da rede. Isso é feito chamando uma função

pertencente ao nível inferior da *task*. Caso não consiga transmitir o *token* após algumas tentativas, o nodo é dado como morto, e esse fato é informado à MSG-TASK. Nesse caso, o *token* será passado para o nodo que normalmente recebia o *token* do nodo morto.

Cada pacote contém, além dos dados ou mensagem, as seguintes informações:

- nodo de origem do pacote,
- nodo destino do pacote,
- tamanho do pacote, em bytes,
- tipo do pacote (dados ou mensagem)

No caso de ser um pacote do tipo dados, contém ainda:

- processo a quem os dados são destinados,
- endereço dentro do espaço de endereçamento do processo onde os dados devem ser colocados,
- segmento desse endereço (dados, código ou pilha).

Quando recebe um pacote do tipo mensagem, a COMM-TASK envia uma mensagem do tipo XMSG_RCVD à MSG-TASK, indicando o endereço de um *buffer* onde o pacote contendo a mensagem recebida está armazenado dentro do espaço de endereçamento da COMM-TASK. Quando a MSG-TASK responder, é sinal a mensagem já foi copiada do *buffer*, que pode ser reutilizado.

No caso de receber um pacote do tipo dados, a COMM-TASK assume que o processo destino dos dados é local e efetua a cópia dos dados para o endereço informado no pacote, através da SYS-TASK. O processo destino certamente será

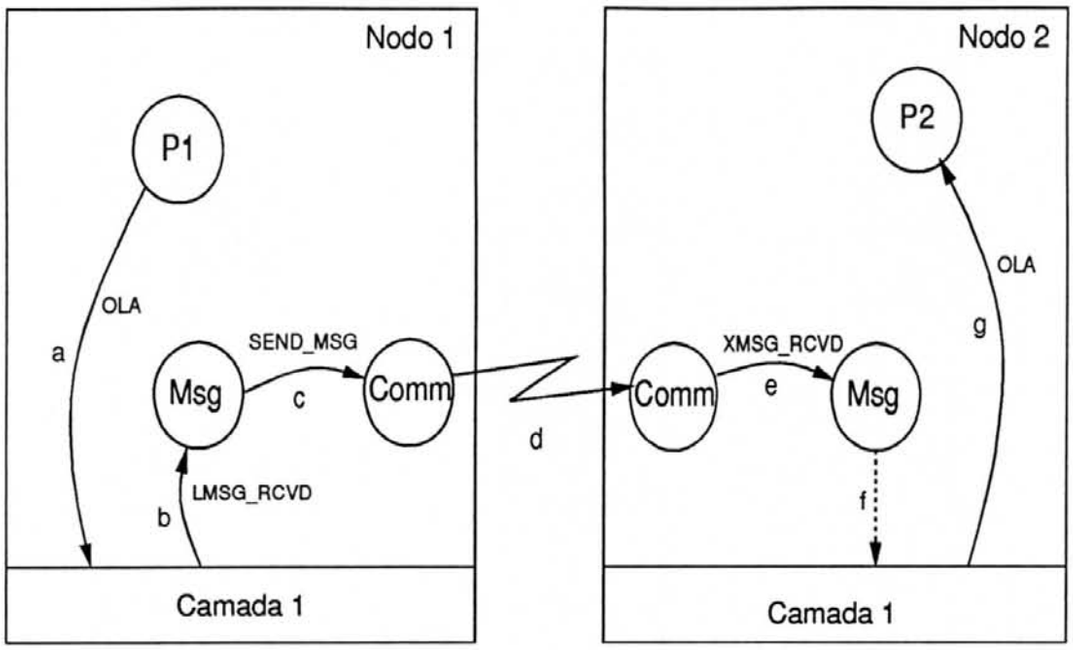


Figura 5.2: Exemplo de troca de mensagem remota

local, porque, para receber dados, esse processo deve ter realizado um pedido a um servidor, e está bloqueado aguardando a resposta do servidor. Conforme visto na seção 5.2, não é permitido migrar processos enquanto estão bloqueados.

## 5.4 Um Exemplo de Troca de Mensagem Remota

Para exemplificar a interação entre as *tasks*, suponhamos que o processo P1 envie uma mensagem do tipo OLA ao processo P2, localizado em um nodo remoto. As operações realizadas paraefetuar a transferência dessa mensagem estão ilustradas na figura 5.2.

a. o processo P1 envia a mensagem ao processo P2, causando uma interrupção de *software* no núcleo do sistema.

b. o núcleo monta uma mensagem do tipo LMSG_RCVD e envia-a à MSG-TASK, informando que o processo P1 deseja enviar uma mensagem

c. a MSG-TASK analisa sua tabela e verifica que o processo P2 está localizado em outro nodo. prepara um pacote de dados com a mensagem, o nodo destino e um número que identifica a mensagem, e envia uma mensagem do tipo SEND_MSG à COMM-TASK, para que envie o pacote ao nodo destino

d. a COMM-TASK, quando receber o *token*, envia o pacote de dados à COMM-TASK do nodo destino

e. ao verificar que o pacote é do tipo mensagem, a COMM-TASK destino envia uma mensagem do tipo XMSG_RCVD à MSG-TASK, informando a chegada de uma mensagem remota

f. após analisar sua tabela e descobrir que o processo destino é local, a MSG-TASK chama uma rotina da camada 1, responsável por enviar mensagens à processos locais

g. a mensagem é finalmente enviada pelo núcleo ao processo P2

No caso de o processo P2 não responder à mensagem dentro de certo período de tempo, a MSG-TASK do nodo do processo P2 enviará uma mensagem do tipo XMSG_ACK à MSG-TASK do nodo origem, informando que a mensagem foi corretamente recebida.

# 6 CONCLUSÃO

Ao longo deste trabalho, foi desenvolvido um mecanismo para suportar a identificação global de processos em um sistema distribuído. Foi também idealizado um algoritmo de comunicação transparente entre processos, que suporta migração de processos e falhas em nodos da rede.

O projeto foi desenvolvido de uma forma modular, que permitiu que o sistema continuasse operacional, mesmo sem a existência de servidores distribuídos. Essa metodologia possibilita a substituição gradual dos servidores centralizados por versões distribuídas, tornando mais simples a tarefa de incorporação de distribuição ao sistema.

A depuração do *software* consumiu parcela considerável do esforço aplicado no desenvolvimento deste trabalho, devido à ausência de ferramentas de depuração adequadas ao ambiente multiprocessado heterogêneo e distribuído. Aliado às dificuldades inerentes à depuração de um sistema distribuído, estão aquelas normalmente associadas ao desenvolvimento do núcleo de um sistema operacional.

Para o futuro, espera-se contar com a substituição do meio físico de comunicação, a fim de obter-se um melhor desempenho nas operações que envolvam transferência de dados entre estações. Os níveis mais altos do protocolo de comunicação continuariam intactos, sendo substituídas apenas as rotinas de controle da interface paralela atualmente utilizada para essas transferências.

Atualmente, não é possível implementar migração de processos, devido a ausência de mecanismos de relocação no *hardware* utilizado. Tal problema seria solucionado com a inclusão de uma unidade de gerenciamento de memória.

O prosseguimento do projeto prevê o desenvolvimento de servidores distribuídos, para que as características distribuídas do núcleo possam ser completamente exploradas.

# ANEXO A-1 COMUNICAÇÃO ENTRE MÁQUINAS

A implementação do protótipo do núcleo do sistema operacional distribuído exigia que as estações possuíssem um meio eficiente para comunicação, o que não era disponível. A solução encontrada para esse problema foi o uso da interface de comunicação paralela existente nas estações. Essa interface provê comunicação unidirecional, visto que foi originalmente projetada para comunicação com impressoras, que são dispositivos destinados unicamente à saída de dados. O protocolo originalmente projetado [BAR90] mostrou-se demasiado lento. Este capítulo descreve as alterações realizadas no *hardware* da interface para tornar bidirecional essa via de comunicação, bem como o novo protocolo desenvolvido para efetivar a comunicação entre as máquinas.

## A-1.1 *Hardware* da Interface Paralela

O circuito da interface paralela do PC é constituído de três portas, duas de saída e uma de entrada[IBM83]. Uma das portas de saída (DATA) possui 8 bits que formam os caracteres que são enviados à impressora. A outra porta de saída é denominada CONTROL e tem 6 bits, quatro utilizados para o envio de sinais de controle à impressora, um para selecionar entre operação por interrupção ou *polling* e um não utilizado. A porta de entrada (STATUS) possui quatro bits, utilizados para que o PC tenha acesso a informações sobre o estado da impressora. Um desses bits de entrada está ligado a uma linha de interrupção do PC, para que a impressora possa avisá-lo quando estiver livre. Existem ainda portas de entrada ligadas às portas de saída, destinadas a teste do circuito e raramente utilizadas. O circuito pode ser observado na figura A-1.1.

O circuito necessário à entrada de dados existe, mas ele está diretamente conectado à saída da própria interface. Isolando-se o circuito de entrada do

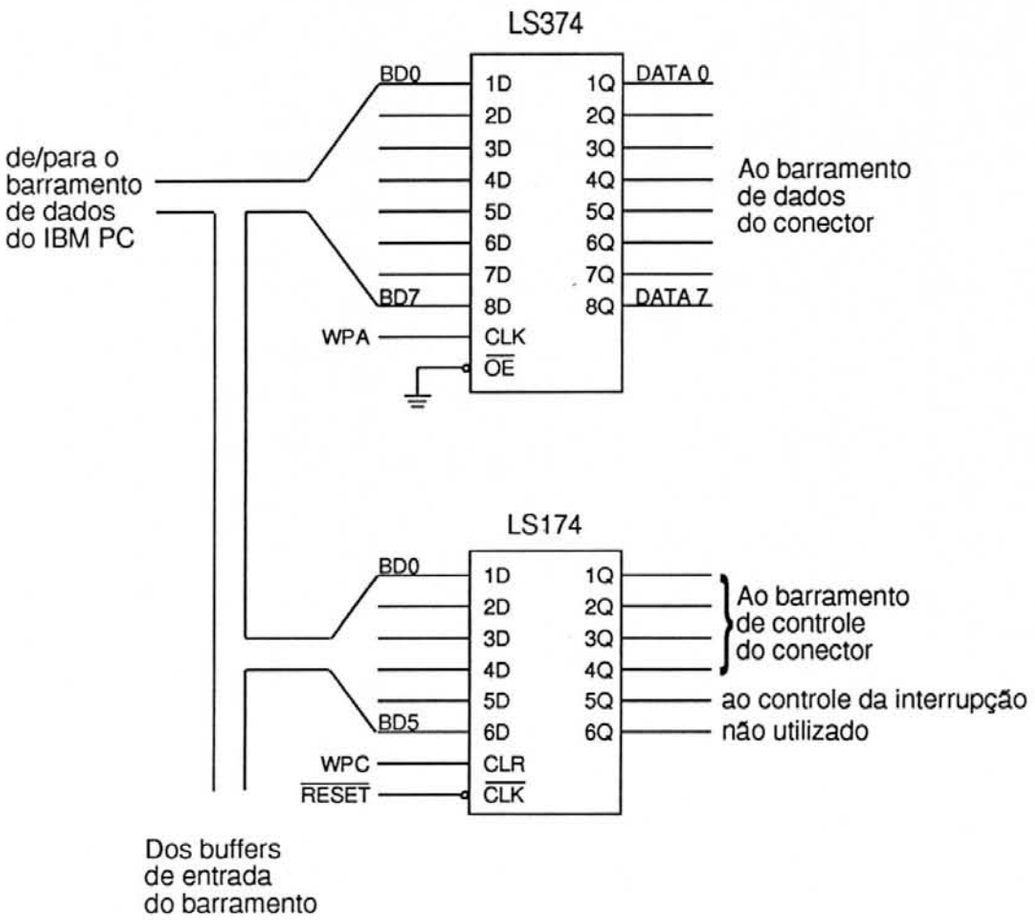


Figura A-1.1: Circuito original da interface paralela

circuito de saída, é possível a leitura dos dados do barramento de forma independente do valor presente na saída. Analisando-se o circuito, verifica-se que a porta de saída possui seu pino de habilitação ($\overline{OE}$—*output enable*) ligado diretamente à terra, o que deixa essa porta permanentemente habilitada. Desconectando-se esse pino da terra e ligando-o ao bit livre da porta de controle conforme a figura A-1.2, é possível desabilitar o circuito de saída. Com esse bit em nível alto o circuito de saída é desabilitado. Essa alteração não afeta o funcionamento normal da interface, pois normalmente é escrito o valor 0 nesse bit. Para não danificar os circuitos de saída da interface, somente uma das interfaces interconectadas poderá estar com sua porta de saída habilitada em um dado instante, e esse controle deve ser realizado pelo *software*.

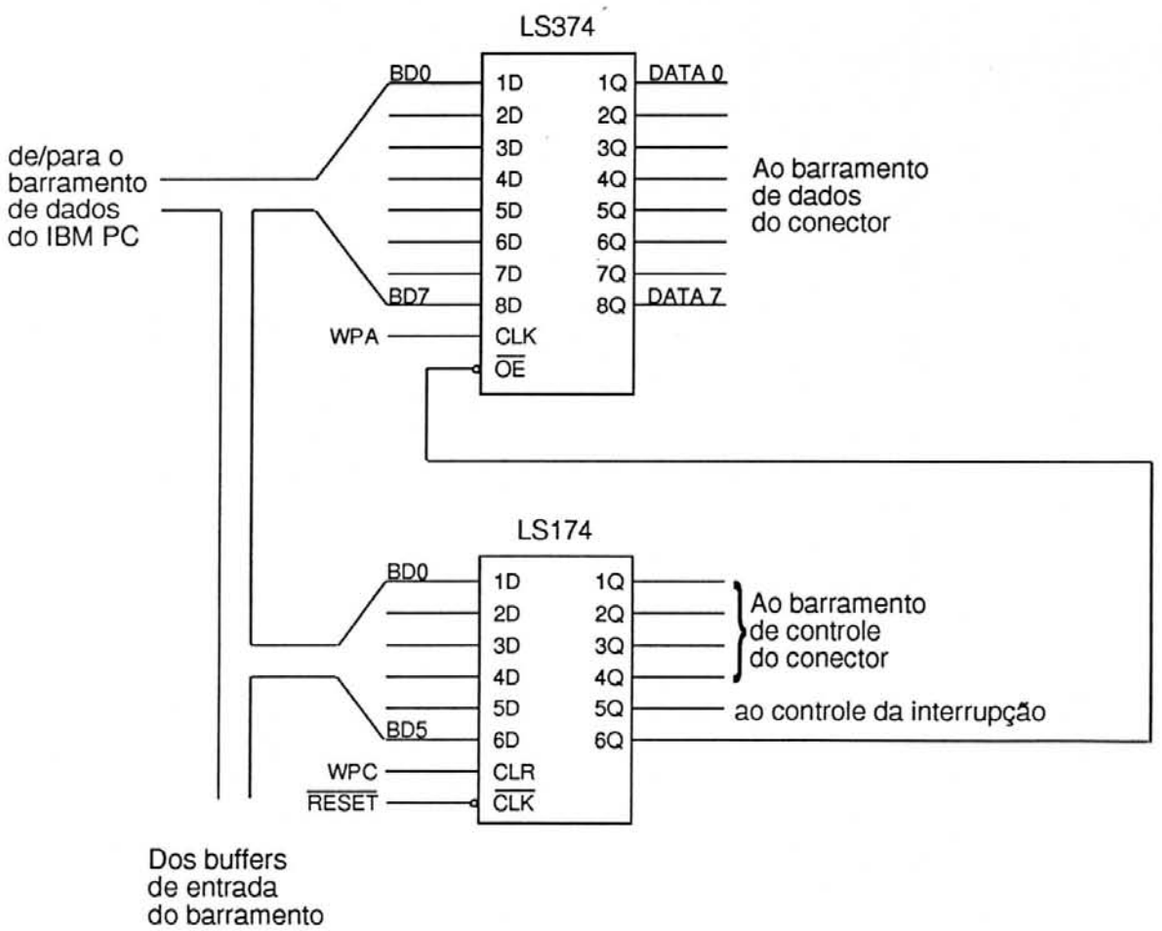


Figura A-1.2: Interface bidirecional

O acesso às portas é realizado por instruções `in` e `out` do processador. Para entrada e saída de dados, é utilizada a porta `DATA`, que normalmente está localizada no endereço `0x378` do espaço de endereçamento de `IO` do 8088. Os bits de controle são acessados pela porta `CONTROL`, que está representada na figura A-1.3, juntamente com a denominação de seus bits. Os bits 4 e 5 dessa porta, denominados `IEN` e `IO`, são somente de escrita, enquanto os quatro bits menos significativos (`TOK`, `ATT`, `ACK` e `STB`) podem também ser lidos. A porta de controle está normalmente localizada no endereço `0x37A`.

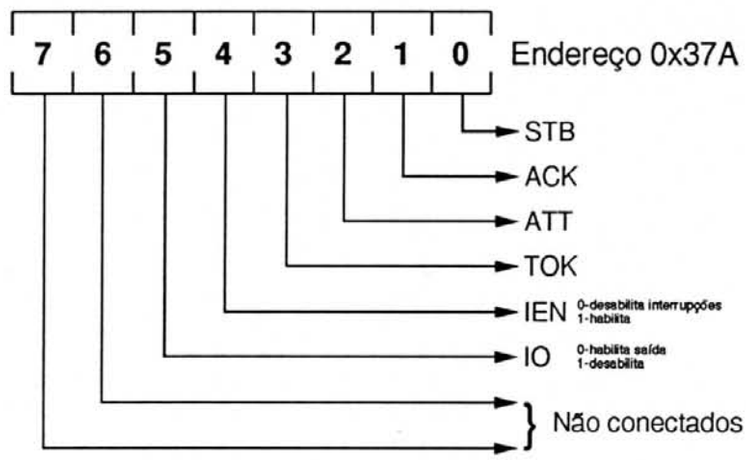


Figura A-1.3: Porta de controle/status

Os *drivers* de saída da porta de controle são configurados como coletor aberto, fazendo com que a leitura de um desses bits corresponda à operação lógica binária AND entre todos os bits correspondentes das máquinas interligadas. Todas as máquinas que não desejam alterar esses bits devem mantê-los em um estado neutro, ou seja, a linha de saída deverá estar em nível alto, que é o valor neutro para a operação AND. Os bits `TOK`, `ACK` e `STB` são invertidos, e o valor que deve ser colocado na porta para que se tenha um estado alto na linha de saída é 0. O bit `ATT` não é invertido, e seu valor neutro é 1.

## A-1.2 O Cabo de Interligação

Para que as várias máquinas possam se comunicar, é necessário um cabo que interligue o barramento de dados e as linhas de controle de suas interfaces paralelas. As portas de saída das máquinas estarão interligadas, podendo ser danificadas caso algum erro no *software* habilite mais de uma ao mesmo tempo. Para evitar dano aos circuitos de saída da interface, existe no cabo uma proteção contra curto-circuitos. Essa proteção é realizada por resistores colocados em série com as linhas de dados. O valor desses resistores foi calculado de forma que eles não atenuem demasiadamente o sinal em condições normais, e limitem a corrente em valores admissíveis pelos circuitos da interface em caso de curto-circuito. As interfaces são interligadas formando um barramento, tanto nas linhas de dados quanto nas de controle, permitindo que quaisquer duas máquinas comuniquem-se diretamente. O cabo está ilustrado na figura A-1.4. Para que seja possível uma estação interromper as demais, a linha de entrada de interrupção do cabo (pino 10) está ligada à linha de saída `ATT` (pino 16). Dessa forma, sempre que houver uma transição de 1 para 0 na linha `ATT`, todas as estações da rede serão interrompidas.

## A-1.3 Descrição do Protocolo

Conforme visto na seção A-1.1, somente uma máquina da rede pode estar com sua porta de saída habilitada em um determinado instante. Dessa forma, somente uma estação pode transmitir dados a cada vez. Isso sugere a necessidade de um *token* para identificar qual estação tem tal direito. A máquina que detém o *token* é a transmissora; todas as demais são receptoras. Esse *token* deve trafegar pela rede, para que todas as estações tenham a chance de transmitir dados para as demais. No protocolo implementado, esse *token* trafega circularmente pela

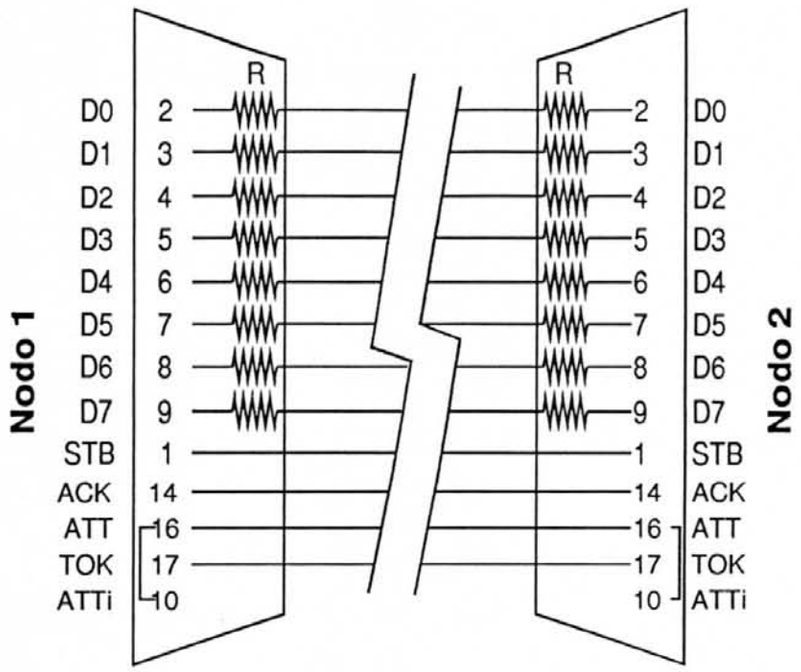


Figura A-1.4: O cabo de interligação

rede, seguindo a ordem dos identificadores das estações. Existem duas formas de comunicação entre estações: transmissão de dados e transmissão de *token*.

Segundo o protocolo descrito a seguir, a linha de controle `ATT` é controlada pelo transmissor, enquanto a linha `ACK` é utilizada pelo receptor para sinalizar o transmissor. As linhas `STB` e `TOK` são controladas ora pelo transmissor ora pelo receptor. Como visto na seção A-1.1, essas linhas devem ser mantidas em um estado neutro por quem não pretende alterar seu valor. A tabela A-1.1 contém o estado em que devem ser mantidos esses bits, pelo transmissor, pelo receptor e pelas demais estações.

Tabela A-1.1: Estados neutros da porta de controle

| *Estação* | IO | IEN | ACK | ATT | STB | TOK |
|---|---|---|---|---|---|---|
| Transmissora | 0 | 0 | 0 | × | × | × |
| Receptora | 1 | 1 | × | 1 | × | × |
| Outras | 1 | 1 | 0 | 1 | 0 | 0 |

Obs.: Os bits marcados por × são alterados durante a transmissão.

### A-1.3.1 Transmissão de Dados

Existem dois tipos distintos de tráfego na rede: mensagens, que são pequenos blocos de algumas dezenas de bytes, e outros dados, que normalmente são da ordem de milhares de bytes. Para evitar sobrecarga na rede enviando grandes pacotes contendo apenas uma mensagem, bem como evitar o *overhead* causado pela divisão de um grande bloco de dados em um número excessivo de pequenos pacotes, foram criados dois tipos de pacotes de dados, de tamanhos distintos. A rede existente é suficientemente confiável na transmissão de pacotes maiores que 1500 bytes. Deixando uma margem de segurança, o maior pacote foi implementado com 1024 bytes, enquanto que o pacote menor possui 256 bytes. A forma de transmissão dos dois tipos de pacotes é bastante semelhante, variam somente na identificação do tipo de comunicação e no número de bytes transmitidos.

Um esquema da transmissão de dados está representado na figura A-1.5. Inicialmente, o transmissor coloca no barramento de dados a identificação da estação com a qual deseja se comunicar (ponto a da figura). A seguir (ponto b), coloca na porta de controle um byte com o tipo de interrupção nos bits TOK e STB, conforme a tabela A-1.2, e com o bit de interrupção ATT ativo (os demais bits, IO, IEN e ACK devem permanecer no estado neutro de transmissão, conforme tabela A-1.1). Nesse momento, as demais estações da rede são interrompidas. A estação que verificar que o valor no barramento de dados coincide com sua identificação lê o byte de controle e avisa o transmissor, ativando o bit ACK de sua porta de controle (ponto c). As demais estações ignoram a interrupção. Caso não obtenha resposta dentro de um determinado período de tempo, o transmissor desiste, acusando erro de *time-out*.

Ao receber a resposta do receptor, o transmissor libera as linhas TOK e STB e desativa ATT para avisar o receptor (ponto d). O receptor então verifica se pode receber os dados do transmissor e transmite essa informação na linha ACK,

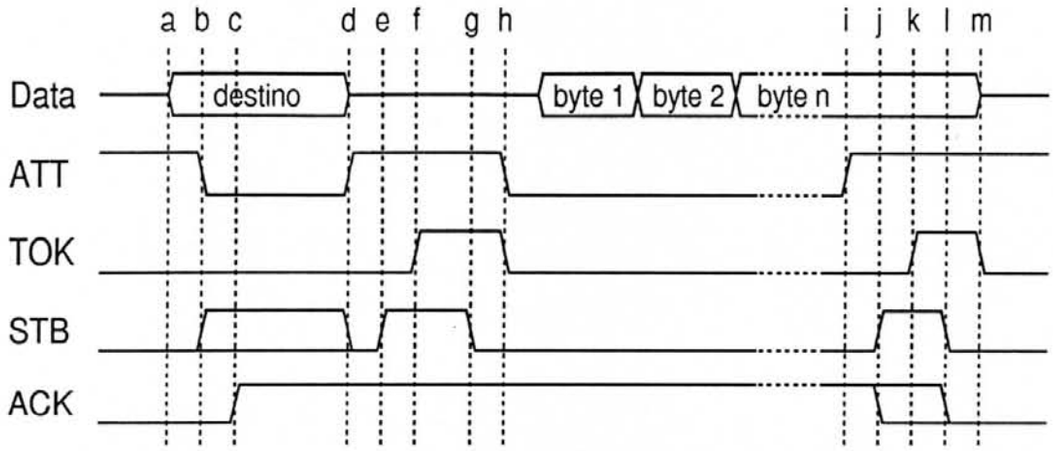


Figura A-1.5: Transmissão de dados

Tabela A-1.2: Tipos de interrupção

| *Tipo de Transmissão* | *Identificação* TOK | STB |
|---|---|---|
| token | 1 | 0 |
| pacote pequeno | 0 | 0 |
| pacote grande | 0 | 1 |

juntamente com STB (ponto e). Quando o transmissor verificar a transição de STB, lerá em ACK a resposta do receptor quanto a sua disponibilidade.

A figura A-1.6 ilustra o caso em que o receptor não pode receber (por não dispor de área de memória livre nesse instante, por exemplo), e aborta a transmissão. Após receber um ACK negativo, o transmissor ativa TOK, avisando o receptor que o ACK foi lido e pedindo para liberar as linhas (ponto f). Logo que vê TOK alto, o receptor coloca suas linhas em estado neutro, desativando STB (ponto g). A comunicação é então finalizada, com o transmissor colocando suas linhas em estado neutro (ponto h). Mais tarde, o transmissor tentará novamente enviar esse mesmo pacote de dados.

Se a resposta do receptor for positiva, o transmissor ativa TOK (figura A-1.5, ponto f), avisando o receptor que recebeu seu sinal ACK. Neste ponto, os dois lados já confirmaram sua intenção de realizar a comunicação. O sincronismo entre transmissor e receptor é realizado por interrupção. O receptor

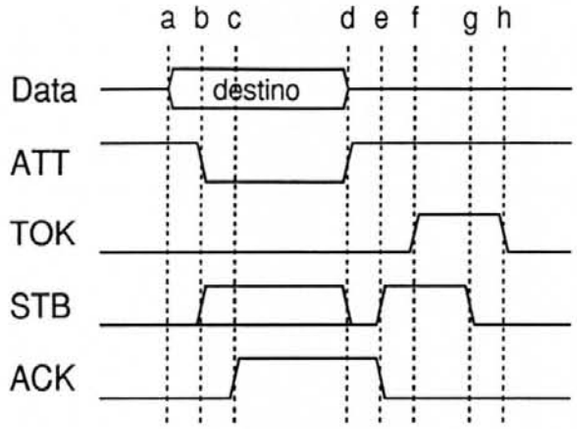


Figura A-1.6: Transmissão de dados abortada pelo receptor

desvia o vetor da interrupção da interface paralela (IRQ 7) para a rotina que lerá os bytes do barramento de dados em sincronia com o transmissor e desabilita as demais interrupções. Caso um ciclo de *refresh* de memória ocorresse durante a transmissão, as estações perderiam o sincronismo. Por essa razão, o *refresh* da memória também é desabilitado pelo receptor neste ponto. Após isso, o receptor desliga `STB` para avisar o transmissor que está pronto para ser interrompido (ponto g) e executa a instrução `hlt`, que paraliza o processador até a ocorrência de uma interrupção de *hardware*.

Quando perceber a inversão de `STB`, o transmissor desabilita seu *refresh*, desativa `TOK`, interrompe o receptor (ponto h) e espera o tempo necessário para que a interrupção seja atendida. Coloca então no barramento de dados os bytes que deseja transmitir, a intervalos fixos, na mesma velocidade que a rotina de interrupção do receptor os lê. A figura A-1.7 mostra o fragmento de código executado pelo transmissor e pelo receptor para a transferência de cada byte, assumindo que o registrador `BX` contém o endereço do próximo byte e o registrador `DX` o endereço da porta de `IO`. Nos computadores utilizados, a uma velocidade do 8 MHz, o tempo de transmissão de cada byte é de aproximadamente 3,5 $\mu$s, o que resulta em uma velocidade de transmissão de 285 kbytes pos segundo, durante a transmissão do pacote. Esse tempo, comparado aos 6 kbytes por segundo conseguidos com o protocolo original mostram a grande vantagem do novo protocolo.

```
mov al,[bx]          in  al,dx
out dx,al            mov [bx],al
inc bx               inc bx
(a) transmissor      (b) receptor
```

Figura A-1.7: Transmissão de um byte

Quando o pacote de dados acabar, cada estação habilitará o *refresh* de sua memória bem como suas interrupções. O transmissor desativa a linha ATT (ponto i). O receptor informa que o pacote foi recebido pelas linhas ACK e STB, com ACK alto se a recepção foi correta (ponto j), o que é respondido pelo transmissor ativando a linha TOK (ponto k). Quando perceber TOK, o receptor coloca suas linhas no estado neutro de recepção (ponto l). Quando verificar isso, o transmissor também coloca suas linhas em estado neutro. A transmissão está completa. Caso tenha sido bem sucedida, o receptor envia uma mensagem à *task* de comunicação indicando a recepção de um novo pacote. Caso contrário, toda a transmissão é ignorada. O transmissor retorna um código indicando se a transmissão foi bem sucedida ou não. Cabe ao nível superior da *task* decidir o que fazer em caso de erro.

Durante a transmissão de um pacote de dados, o transmissor interrompe o receptor duas vezes, uma para iniciar contato e outra para realizar o sincronismo. Como todos os receptores estão com suas interrupções habilitadas, todos serão interrompidos. É necessário garantir que somente o receptor desejado reconhecerá as interrupções. A primeira interrupção é reconhecida porque a identificação do receptor está no barramento de dados. No entanto, algum receptor poderia estar com suas interrupções desabilitadas e somente atender a interrupção mais tarde, quando a transmissão de dados para o receptor correto já estivesse em andamento. Neste caso, algum dado intermediário presente no barramento poderia ser confundido com a identificação do receptor. Para evitar isso, a linha de interrupção ATT é desligada logo após a interrupção haver sido reconhecida pelo receptor correto e antes de ser retirada a identificação do receptor do

barramento de dados (ponto d da figura A-1.5). Os receptores, após reconhecerem sua identificação no barramento de dados, devem portanto verificar o estado da linha ATT e ignorar a interrupção se esta não estiver ativa. A segunda interrupção pode ser ignorada porque, durante toda a transmissão do bloco de dados, a linha ACK é mantida alta pelo receptor. Ao serem interrompidos, os receptores avaliam o estado das linhas ATT e ACK, e ignoram a interrupção caso algum deles esteja em nível alto.

A detecção de erros de transmissão é feita pelo envio, ao final do pacote de dados, de um byte de controle contendo a operação lógica ou- exclusivo entre alguns bytes do pacote. O receptor também calcula o valor desse byte e o compara com o recebido do transmissor. Caso sejam diferentes, houve algum erro durante a transmissão, e o bloco de dados é ignorado. Para maior segurança, esse byte de controle é transmitido duas vezes, a segunda com todos seus bits invertidos. Os erros de transmissão verificados durante testes com o sistema deveram-se à perda de sincronismo entre as estações causada pelo atendimento de pedidos pendentes de DMA durante a transmissão, e foram todos detectados por este esquema.

### A-1.3.2 Envio do *token*

A transmissão do *token* é bem mais simples que a transmissão de dados, conforme pode ser visto na figura A-1.8. Inicialmente, o transmissor coloca a identificação do receptor no barramento de dados (ponto a da figura). A seguir indica transmissão de *token* nas linhas TOK e STB, conforme a tabela A-1.2 e interrompe o receptor (ponto b). Após reconhecer a interrupção e verificar, pelo estado das linhas TOK e STB, que se trata de transmissão de *token*, o receptor ativa o bit ACK (ponto c). Quando receber o ACK do receptor, o transmissor coloca sua porta de controle na condição neutra de recepção—tornou-se receptor (ponto d). O receptor, ao perceber isso (a linha ATT vai a nível alto) transforma-se no próximo transmissor: coloca a condição neutra de transmissão na sua porta de controle

(ponto e) e envia uma mensagem à *task* de comunicação informando-lhe a posse do *token*.

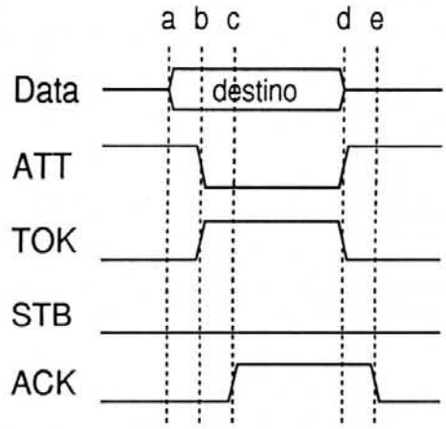


Figura A-1.8: Transmissão do *token*

### A-1.3.3 Saída de uma estação da rede

Eventualmente, de forma intencional, por falha ou por acidente, uma estação pode desligar-se da rede. As demais estações, a não ser por eventos diretamente relacionados com a estação que se desligou, devem continuar normalmente sua comunicação. Existem duas situações possíveis, dependendo da situação da estação no momento de sua falha, se estava como receptora ou transmissora.

No caso de a estação a sair da rede ser no momento uma receptora, a solução para o problema é simples. Quando a estação predecessora da estação que saiu tentar por algumas vezes lhe transmitir o *token* sem obter sucesso, passa a considerá-la desligada da rede. A partir de então, não será mais tentado o envio do *token* para essa estação, e a próxima estação passa a ser considerada sua sucessora. O nível superior da *task* de comunicação é informado, e cabe ao mesmo decidir o que fazer com os pacotes de dados pendentes destinadas à estação que se desligou. É também o nível superior da *task* que informa outras *tasks* do sistema, como as responsáveis pelo roteamento de mensagens e dados entre as estações.

Por outro lado, se a estação vítima da falha estiver de posse do *token*, o mesmo será perdido. Nesse caso, é necessário que outra estação da rede recrie o *token*, para que a comunicação entre as estações restantes continue. Uma estação cria o *token* sempre que verificar que a rede está sem atividade por um período de tempo proporcional ao número de sua identificação. A estação de número 1 criará o *token* após n segundos sem tráfego na rede; a estação de número 2, após 2n segundos, e assim por diante. O tráfego na rede é reconhecido através das interrupções para envio de dados ou *token*, que são sentidas por todas as estações da rede.

### A-1.3.4 Entrada de uma estação na rede

Para que uma estação possa se comunicar com as demais estações da rede, é necessário que essa estação receba o *token* de sua estação predecessora. Entretanto, enquanto a estação que deseja entrar na rede não obtiver o *token*, não tem meios de informar sua predecessora que tem intenções de participar da rede. É necessário uma forma alternativa de ingresso de novas estações na rede.

Cada estação é responsável pela entrada das estações com número de identificação superior ao seu e inferior ao da próxima estação atualmente ativa na rede. Após um determinado número de transmissões do *token*, cada estação tenta enviar o *token* para uma das estações nesse intervalo. Se a estação responder, passa a integrar a lista de estações ativas, passando a receber o *token* regularmente. A partir desse momento, as camadas superiores do sistema operacional podem se comunicar com as demais estações, para oferecer os recursos da nova estação ao sistema, efetivando sua integração à rede.

# BIBLIOGRAFIA

[ART87] ARTSY, Yeshayahu; CHANG, Hung-Yang; FINKEL, Raphael. Interprocess Communication in Charlotte. **IEEE Software.** New York, v.4, n.1, p.22–28, 1987

[BAR90] BARCELLOS, Antônio M. P.; STEIN, Benhur de O.; LUZ, Marcos V. I.; BELMONTE Filho, Valdir R. **DIX Projeto e Implementação de um Sistema Operacional Distribuído para uma Rede de Estações de Trabalho** Porto Alegre: CPGCC da UFRGS, Out. 1990. 77p. (Projeto de Pesquisa)

[BAR90a] BARCELLOS, Antônio M. P.; LUZ, Marcos V. I.; STEIN, Benhur de O.; BELMONTE Filho, Valdir R. DIX: Um Sistema Operacional Distribuído para Estações de Trabalho Multiprocessadoras Heterogêneas. In: SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES E PROCESSAMENTO PARALELO, 3., Nov. 7–9, 1990, Rio de Janeiro. **Anais...** Rio de Janeiro: SBC, 1990. p.138–150.

[BEL90] BELMONTE Filho, Valdir R. **Especificação do Sistema Operacional Multiprocessado MINIXM para a Máquina M3P.** Porto Alegre: CPGCC da UFRGS, 1990. (Trabalho Individual n.148).

[BEL92] BELMONTE Filho, Valdir R. **Gerência de Processos em Sistemas Distribuídos Tolerantes a Falhas.** Porto Alegre: CPGCC da UFRGS, 1992. Dissertação de Mestrado a ser publicada.

[CAR89] CARRERAS, Marcelo S. **Projeto de Implementação da Arquitetura M3P.** Porto Alegre: CPGCC da UFRGS, 1989. (Trabalho Individual).

[CHA90] CHANDRAS, Rajan G. Distributed Message Passing Operating Systems. **Operating Systems Review,** New York, v.24, n.1, p.7–17, Jan. 1990.

[CHE83] CHERITON, David R.; ZWAENEPOEL, Willy. The Distributed V Kernel and its Performance for Diskless Workstations. **Operating Systems Review,** New York, v.17, n.5, p.129–140, 1983. Trabalho apresentado no Symposium on Operating Systems Principles, 9., Bretton Woods, Oct. 10–13, 1983.

[CHE86] CHERITON, David R. Request-Response and Multicast Interprocess Communication in the V Kernel. In: INTERNATIONAL SEMINAR ON NETWORKING IN OPEN SYSTEMS, Aug. 18–22, 1986, Oberlech. **Proceedings...** Berlin: Springer-Verlag, 1987. 441p. p.296–312. (Lecture Notes in Computer Science, 248).

[CHE88] CHERITON, David R. The V Distributed System. **Communications of the ACM,** New York, v.31, n.3, p.314–333, Mar. 1988.

[CRI88] CRICHLOW, Joel M. **An Introduction to Distributed and Parallel Computing.** Hertfordshire: Prentice Hall, 1988. 209 p.

[FIN89] FINKEL, Raphael A.; SCOTT, Michael L.; ARTSY Yeshayahu; CHANG, Hung-Yang. Experience with Charlotte: Simplicity and Function in a Distributed Operating System. **IEEE Transactions on Software Engineering,** New York, v.15, n.6, Jun. 1989.

[GEI90] GEIHS, Kurt; HOLLBERG, Ulf. Retrospective on DACNOS. **Communications of the ACM,** New York, v.33, n.4, p.439–448, Apr. 1990.

[IBM83] IBM: **IBM Personal Computer Technical Reference Manual** International Business Machines, Apr. 1983. 302p.

[KER78] KERNIGHAN, Brian W.; RITCHIE, Dennis M. **The C Programming Language.** Englewood Cliffs, Prentice Hall, 1978.

[KUT84] KUTTI, Swamy. Why a Distributed Kernel? **Operating Systems Review,** New York, v.18, n.4, p.5–11, Oct. 1984.

[MUL86] MULLENDER, Sape J.; TANENBAUM, Andrew S. The Design of a Capability-Based Operating System. **The Computer Journal,** v.29, n.4, p.286–299, 1986.

[MUL87] MULLENDER, Sape J. Process Management in a Distributed Operating System. In: INTERNATIONAL WORKSHOP ON EXPERIENCES WITH DISTRIBUTED SYSTEMS, Sep. 28–30, 1987, Kaiserslautern. **Proceedings...** Berlin: Springer-Verlag, 1988. 292p. p.38–51. (Lecture Notes in Computer Science, 309).

[MUL90] MULLENDER, Sape J.; VAN ROSSUM, Guido; TANENBAUM, Andrew S.; VAN RENESSE, Robbert; VAN STAVEREN, Hans. Amoeba a Distributed Operating System for the 1990s. **Computer,** New York, v.23, n.5, p.44–53, May 1990.

[NAV90] NAVAUX, Philippe O. A. M3P: Máquina e Sistema Operacional Multiprocessadores. In: CONGRESSO NACIONAL DE INFORMÁTICA SUCESU, 12., 1990, Rio de Janeiro. **Anais...** Rio de Janeiro: SUCESU, 1990. Edição em disquetes.

[POW83] POWELL, Michael L.; MILLER, Barton P. Process Migration in DEMOS/MP. **Operating Systems Review,** New York, v.17, n.5, p.110–119, Oct. 1983.

[RIT74] RITCHIE, Dennis M.; THOMPSON, K. The UNIX Time-Sharing System. **Communications of the ACM,** New York, v.17, n.7, p.365–375, July 1974.

[SHE86] SHELTZER, Alan B.; POPEK, Gerald J. Internet Locus: Extending Transparency to an Internet Environment. **IEEE Transactions on Software Engineering,** New York, v.12, n.11, p.1067–1075, Nov. 1986.

[SMI88] SMITH, Jonathan M. A Survey of Process Migration Mechanisms. **Operating Systems Review,** New York, v.22, n.3, p.28–40, Jul. 1988.

[STA84] STANKOVIC, John A. A Perspective on Distributed Computer Systems. **IEEE Transactions on Computers,** New York, v.33, n.12, p.1102–1115, Dec. 1984.

[STE89] STEIN, Benhur de Oliveira. **Desenvolvimento do Módulo Ouvidor da Máquina M3P—Software** Porto Alegre: CPGCC da UFRGS, 1989. 56p. (Trabalho Individual n.132).

[SUN90] SUN Microsystems. **Network Programming Guide.** Palo Alto: Sun Microsystems, 1990. 346p.

[SVO85] SVOBODOVA, Liba. Workshop Summary — Operating Systems in Computer Networks. **Operating Systems Review,** New York, v.19, n.2, p.6–37, Apr. 1985.

[TAN81] TANENBAUM, Andrew S.; MULLENDER, Sape J An Overview of the Amoeba Distributed Operating System. **Operating Systems Review**, New York, v.15, n.3, p.51–64, Jul. 1981.

[TAN85] TANENBAUM, Andrew S. Distributed Operating Systems. **Computing Surveys,** New York, v.17, n.4, p.419–470, Dec. 1985.

[TAN87] TANENBAUM, Andrew S. **Operating Systems: Design and Implementation.** Englewood Cliffs: Prentice Hall, 1987.

[TAN87] TANENBAUM, Andrew S. A Unix Clone with Source Code for Operating Systems Courses. **Operating Systems Review**, New York, v.21, n.1, p20–29, Jan. 1987.

[TAN90] TANENBAUM, Andrew S.; VAN RENESSE, Robbert; VAN STAVEREN, Hans; SHARP, Gregory J., et al. Amoeba System. **Communications of the ACM**, New York, v.33, n.12, p.46–63, Dec. 1990.

**Informática**
*UFRGS*

"Projeto do Núcleo de um Sistema Operacional Distribuído".

Dissertação apresentada aos Srs.:

Prof. Dr. Cláudio Fernando Resin Geyer.

Prof. Dr. Philippe O. A. Navaux

Prof. Dr. Raul Fernando Weber

Prof. Thadeu Botteri Corso (UFSC)

Vista e permitida a impressão.
Porto Alegre, 18/11/92.

Prof. Dr. Philippe O. A. Navaux,
Orientador.

Prof. Dr. Ricardo A. da L. Reis,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.