# PROJETO DE UM SUPORTE PARA SISTEMA OPERACIONAL DISTRIBUÍDO COM RECONFIGURAÇÃO DINÂMICA DE PROCESSOS

Durval Makoto Akamatu

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO

Aprovado por:

Profª Sueli B. Teixeira Mendes, Ph.D
( Presidente )

Prof. Edil S.T. Fernandes, Ph.D

Prof. Valmir Carneiro Barbosa, Ph.D

Prof. Claudio Kirner, D.Sc.

Prof. José Hiroki Saito, Dr.

RIO DE JANEIRO, RJ - BRASIL
MAIO DE 1991

AKAMATU, DURVAL MAKOTO

Projeto de um Suporte para Sistema Operacional Distribuído com Reconfiguração Dinâmica de Processos [Rio de Janeiro] 1991.

XIII, 249 p. 29,7 cm (COPPE/UFRJ, D.SC., Engenharia de Sistemas e Computação (1991).

Tese - Universidade Federal do Rio de Janeiro, COPPE.

1. Sistemas Distribuídos I. COPPE/UFRJ II. Título (série).

Esta tese é dedicada:

- À minha esposa e filhos: Joanita, Selma, Simone, Saulo e Silmar;
- À minha mãe: Mizuho Yato;
- Aos meus irmãos: Cezar, Lina e Ruvens
- Ao Professor Dante A.O. Martinelli da USP, Campus de São Carlos

# AGRADECIMENTOS

Á Professora Sueli Mendes, pela dedicação na orientação, incentivo, apoio e muita paciência para comigo, desde o início do desenvolvimento do trabalho.

Ao Professor Claudio Kirner, pela sua contribuição como co-orientador que, também, sempre nas horas mais difíceis não faltou com seu apoio desde o início do trabalho.

Ao Professor Luis C. Trevelin, que juntos iniciamos a nossa caminhada para dar mais um passo na capacitação docente, que sempre me incentivou e trocou idéias sobre assuntos relacionados ao trabalho.

À Professora Marina T.P. Vieira, que também iniciou seu trabalho na mesma época e que junto com o Professor Trevelin me fez crescer mais como pessoa nesses últimos anos.

À UFSCar e em especial ao Departamento de Computação pela concessão de meu afastamento das atividades docentes e administrativas durante o período inicial deste trabalho.

Aos professores, técnicos e o pessoal administrativo do Departamento de Computação da UFSCar, que de uma forma ou de outra, contribuíram para o bom desenvolvimento do trabalho.

À CAPES, pelo auxílio concedido.

Aos Professores Edil S.T. Fernandes e Lídia Segre do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ que sempre se dispuseram a esclarecer minhas dúvidas, principalmente na fase inicial.

À Denise e Cláudia, secretárias do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, pela amizade e auxílio dado nas questões burocráticas.

Ao Orestes Cezetti, aluno de pós-graduação do Departamento de Computação da UFSCar, pelo estudo em conjunto e implementação de boa parte do trabalho prático.

À minha família pela paciência e compreensão que tiveram com a minha ausência física e espiritual de seus convívios, principalmente durante a fase mais intensa deste trabalho.

Aos Professores Dante A.O. Martinelli e Julieta P. Martinelli, pelo incentivo e apoio dado para que eu pudesse conciliar o trabalho e o estudo, durante a minha formação de engenheiro civil.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências (D.Sc.)

# PROJETO DE UM SUPORTE PARA SISTEMAS OPERACIONAIS DISTRIBUÍDOS COM RECONFIGURAÇÃO DINÂMICA DE PROCESSOS

Durval Makoto Akamatu

MAIO, 1991

Orientadora: Sueli B.T. Mendes
Programa : Engenharia de Sistemas e Computação

Este trabalho trata dos aspectos de tolerância a falhas em Sistemas Operacionais Distribuídos, abordando essencialmente falhas provenientes de defeitos nos computadores componentes do Sistema Distribuído e de falhas no subsistema de comunicação.

Assim, nesse contexto, buscam-se apresentar os elementos teóricos e práticos para a implementação de um suporte para o desenvolvimento de Sistemas Operacionais Distribuídos com a característica de serem tolerantes a esses tipos de falhas. Para tanto, descreve-se e discute-se um núcleo para Sistemas Operacionais Distribuídos que oferece um mecanismo para comunicação e sincronização entre processos, baseado no conceito de portos, que é tolerante a falhas e mecanismos para recuperação e migração de processos, para superar defeitos apresentados nos computadores do sistema.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

# PROJECT OF A SUPPORT FOR DISTRIBUTED OPERATING SYSTEMS WITH DINAMIC RECONFIGURATION OF PROCESSES

Durval Makoto Akamatu

MAY, 1991

Adviser: Sueli Mendes
Departament: Engenharia de Sistemas e Computação

This Work considers aspects of fault tolerance in Distributed Operating Systems, particulary the faults in computers of the Distributed System as well as failures in the communication subsystem.

In this context, it is presented the theorical and practical elements to the implementation of a framework to the development of a Fault-tolerant Distributed Operating Systems. This work discribes and discusses a Distributed Operating Systems Kernel wich provides a mechanism for processes communication and synchronization, based on the concepts of ports, which are fault tolerance, and mechanisms for the processes recovery and migration, to treat the faults presented by the system computers.

# ÍNDICE

# ÍNDICE DE FIGURAS

CAPÍTULO V



CAPÍTULO VI

# CAPÍTULO I
# INTRODUÇÃO

## I.1 CONTEXTO GERAL DA PESQUISA

Os Sistemas Distribuídos, em seu amplo sentido, têm sido objeto de pesquisa há pelo menos duas décadas que, sob influência dos avanços significativos obtidos, cresceram em importância e utilização. Entretanto, ainda não se tem um amplo domínio da construção de "softwares" distribuídos, ou melhor, de Sistemas Operacionais Distribuídos, como se tem para sistemas centralizados. Deve-se isso aos muitos novos problemas serem bem mais complexos. Problemas esses, implicitamente levantados pelos objetivos gerais desejados para sistemas desse tipo que são, em ordem:

- que os múltiplos processadores independentes sejam controlados por um único Sistema Operacional de forma integrada;
- que este ambiente seja transparente aos usuários; e
- seja tolerante a falhas.

e principalmente, que os índices: eficiência e confiabilidade, sejam no mínimo muito próximos aos de um sistema centralizado de porte equivalente e que sua disponibilidade seja maior.

Atualmente se está ciente de que muitos conceitos e técnicas usados para a construção de "software" para sistemas centralizados, podem ser perfeitamente aplicados nos Sistemas Distribuídos, podendo diferenciar um pouco nas técnicas. Por exemplo, pode-se citar a estruturação em camadas hierárquicas, a Programação N-Versão, etc. Também sabe-se que, para alguns dos novos problemas, eles não são adequados, exigindo outros mais específicos como, por exemplo, para resolver o problema da obtenção do estado global do sistema, ou ainda, para a construção de um mecanismo eficiente para recuperação de processos em tempo de execução, etc.

Diante disso, as pesquisas têm-se concentrado na elaboração de soluções eficientes para os novos problemas. Dessas soluções, muitas continuam ainda no plano teórico e das colocadas em práticas experimentais, muitas são direcionadas para sistemas cujos "hardwares" foram especificamente elaborados de forma que os requisitos desejados ( confiabilidade, disponibilidade e desempenho ), fossem considerados satisfatórios. Outras experimentações, ignoraram o desempenho, por exemplo, e procuram averiguar a eficácia de determinada técnica como uma solução para um problema específico e assim por diante. Apenas a título de ilustração, cita-se alguns dos problemas: balanceamento dinâmico de cargas nos componentes de processamento do sistema, migração de processos e recuperação de sistemas, não querendo dizer com isso que não existam propostas de soluções para eles.

Dessa maneira, a exigência ou a necessidade dessa quantidade extra de "software" para que um Sistema Distribuído possa ser visto como uma única máquina, com as vantagens desejadas em relação aos centralizados, faz com que os Sistemas Operacionais Distribuídos para uso geral se tornem, a princípio, bem maiores e mais complexos e, com isso, continuando sendo um desafio.

## I.2 OBJETIVOS DO TRABALHO

A técnica muito eficiente, difundida e usada para o desenvolvimento de "software" desse tipo e porte, é a da estruturação em camadas hierárquicas. Nessa estruturação, o "software" básico,

consistindo das camadas de mais baixo nível que, obviamente, inclui aquela que faz interface com o "hardware" subjacente, é considerada a parte mais importante para o desenvolvimento de Sistemas Operacionais. Isso, porque ele se constitui na máquina virtual responsável pela execução das facilidades oferecidas para a implementação das camadas de "software" de nível mais alto ( ou máquinas virtuais de níveis mais altos ).

Dessas várias camadas de "software", o presente trabalho se preocupou no projeto do "software" básico, ou suporte básico. Mais explicitamente, o trabalho atuou no suporte para o desenvolvimento de Sistemas Operacionais Distribuídos que tenham a característica de serem reconfiguráveis dinamicamente.

Com esse objetivo, procurou-se desenvolver o referido suporte de forma que ele se constituísse em uma máquina virtual básica tolerante a falhas da máquina subjacente e que, assim, ofereça facilidades confiáveis para o desenvolvimento de Sistema Operacional Distribuído com aquela característica.

Dentre as várias facilidades que o suporte deve oferecer para o desenvolvimento de Sistemas Operacionais Distribuídos para uso geral, com a característica de ser reconfigurável dinamicamente, abordaram-se as seguintes:

- para comunicação e sincronização entre processos;
- para recuperação de processos; e
- para a migração de processos.

Para isso, utilizaram-se como referências básicas os trabalhos de SHRIVASTAVA ( 1983 ), PANZIERI e SHRIVASTAVA ( 1985, 1988 ), POWELL e MILLER ( 1983 ), BORG et alii ( 1983 ) e POWELL e PRESOTTO ( 1983 ).

A facilidade para comunicação e sincronização entre processos baseou-se no conceito de portos locais. Por questões de confiabilidade, incorporou-se nesse, mecanismos para tolerar falhas de comunicação decorrentes de colapso de processador, perda da mensagem de resposta, etc., que podem dar surgimento a outros eventos indesejáveis denominados de órfãos ( mensagens e computações órfãs ). Tais eventos indesejáveis se não forem imediatamente tratados, no mínimo desperdiçarão tempo de processamento ( SHRIVASTAVA, 1983; PANZIERI e SHRIVASTAVA, 1985, 1988 ). Diante disso, tal mecanismo para comunicação e sincronização entre processos foi denominado de

mecanismo para comunicação e sincronização entre processos baseado em portos, com suporte para detecção e eliminação de órfãos.

A facilidade, ou melhor, o mecanismo para recuperação de processos é o suporte que terá a função principal de tolerar colapsos de nodos operadores, recuperando os processos ali residentes para um outro nodo e, com isso, impedindo que o sistema entre em colapso. As outras funções serão as de dar o suporte para o tratamento de órfãos, mais especificamente computações órfãs.

Por último, a facilidade para migração de processos é essencial tanto como mecanismo suporte para recuperação de processos ( recuperação de um processo residente em um nodo para um outro é acompanhada de uma migração ), como para permitir que o sistema possa com maior flexibilidade realizar as tarefas de configuração inicial e reconfiguração dinâmica por questões de recuperação ou de balanceamento dinâmico de carga.

Visto que, o assunto do trabalho refere-se a análise e desenvolvimento de "software" básico para Sistema Operacional Distribuído com característica de ser tolerante a falhas, faz-se uma rápida revisão sobre caracterização de Sistemas Distribuídos, seguido de um estudo geral sobre tolerância a falhas e uma revisão sobre mecanismos para comunicação e sincronização entre processos baseado no conceito de troca de mensagens. Para finalizar, apresentam-se o desenvolvimento e a descrição de um suporte para construção de Sistemas Operacionais Distribuídos com Reconfiguração Dinâmica de Processos.

## I.3 ORGANIZAÇÃO DO TRABALHO

O trabalho está organizado em oito capítulos conforme segue.

O capítulo II, "Sistemas Distribuídos e Tolerância a Falhas", apresenta a caracterização de Sistemas Distribuídos, os aspectos de (ios e conceitos de sistemas de computação tolerante a falhas.

No capítulo III, "Estruturação de Sistemas Distribuídos Tolerantes a Falhas", são levantados e discutidos os conceitos e técnicas para estruturação de "software" para sistemas de grande porte, visando diminuir a complexidade, principalmente, para Sistemas Distribuídos que precisam apresentar um alto grau de confiabilidade, isto é, que sejam

tolerantes a falhas. Também, são discutidos os mecanismos encontrados na literatura para tratar falhas previstas ou não, que foram propostos, visando única e exclusivamente contribuir com soluções para os problemas de confiabilidade, levantados no capítulo anterior. Dessa forma, praticamente todas as propostas foram elaboradas, de modo que elas fossem aplicáveis para qualquer tipo de sistema que necessita tolerar falhas. A importância da discussão dessas propostas no contexto do presente trabalho está, não só em apresentar um levantamento bibliográfico a respeito de confiabilidade, mas também porque elas serviram e continuam servindo como base para a elaboração de soluções alternativas. Isso vem contribuindo para os novos problemas impostos pelo avanço tecnológico que oferecem recursos para a construção de máquinas de arquitetura das mais variadas possíveis, dentre elas, aquela em pauta, os Sistemas Distribuídos. Finaliza-se o capítulo, apresentando a estruturação de Sistemas Distribuídos Tolerantes a Falhas através de componentes idealizados tolerante a falhas, proposta por RANDELL ( 1984 ).

No capítulo IV, "Recuperação de Sistemas Distribuídos", são abordados os mecanismos propostos para o tratamento de falhas previstas ou não, que consideram o sistema como um todo. Isso é, com a preocupação de que muitas das falhas podem ser detectadas após passado um tempo e cujos confinamentos não foram possíveis e, devido a isso, elas poderiam já ter contaminado o sistema. Assim, o mecanismo para tratamento da falha detectada deve levar em consideração essa hipótese; uma falha desse tipo é o colapso de um processo. Enfim, discutem-se os mecanismos que para tratarem devidamente as falhas detectadas, colocam o sistema em um estado anterior consistente, que poderia ou não realmente ter existido. Em seguida, analisam-se as propostas que consideram, como fator importante para a recuperação de sistema, as trocas de mensagens entre os seus processos que, para isso, impõem que estes sejam determinísticos. Completa-se o presente capítulo com a análise dos mecanismos para migração de processos e o problema da presença de órfãos no sistema. Ambos importantes para os objetivos diretos deste trabalho.

O capítulo V, "Sistemas Operacionais Distribuídos", objetivou a discussão das funções de um núcleo para Sistemas Operacionais Distribuídos, formas de implementação e os mecanismos para comunicação e sincronização entre processos baseado em troca de mensagens. Também, apresenta a análise de alguns dos Sistemas Operacionais

Distribuídos encontrados na literatura, quanto às suas estruturas, tipos de mecanismos de comunicação e sincronização entre processos utilizados e quanto às suas características de tolerância a falhas. Por fim, de um modo geral, discute-se brevemente a importância das linguagens de programação para a implementação de Sistemas Operacionais.

No capítulo VI, "Desenvolvimento de um Suporte para Construção de Sistemas Distribuídos Tolerantes a Falhas", apresenta-se a contribuição principal da tese que, basicamente, consiste de um mecanismo para comunicação e sincronização entre processos baseado no conceito de portos locais, com suporte para detecção e eliminação de órfãos. Os mecanismos para recuperação e migração de processos descritos, complementam a contribuição.

O capítulo VII, "Implementação de um Suporte para Sistema Operacional Distribuído com Reconfiguração Dinâmica de Processos", apresenta as primitivas do suporte descrito no capítulo anterior e aquelas para o gerenciamento de processos, necessárias para uma descrição concisa de um protótipo de Sistema Operacional Distribuído com Reconfiguração Dinâmica de Processos, para fins de exemplificação.

As conclusões finais do trabalho e sugestões para futuras pesquisas são descritas no capítulo VIII - "Conclusões".

Por último, são listadas as referências bibliográficas consultadas para o desenvolvimento do trabalho.

# CAPÍTULO II

## SISTEMAS DISTRIBUÍDOS E TOLERÂNCIA A FALHAS

O termo "Sistema Distribuído" continua ainda sem um consenso sobre a sua definição que o caracterize de forma única. Existem várias definições que diferem em alguns fatores determinantes. Isso porque, um Sistema de Computação pode apresentar vários elementos de "hardware" e de "software" distribuídos e, muitas vezes, a distribuição de qualquer um desses elementos, por exemplo os dados, pode caracterizá-lo como tal. Nota-se, entretanto, que a grande maioria desses fatores são comuns nessas definições ( veja KIRNER, 1986; KIRNER e MENDES, 1988 ).

Objetivando identificar os fatores determinantes de Sistemas Distribuídos, KIRNER ( 1986 ) e KIRNER e MENDES ( 1988 ), fazem uma análise detalhada das características desses sistemas, no sentido de enquadrá-los em uma área mais ampla, de acordo com seus fatores determinantes.

Assim, descreve-se uma síntese dessas características e delinea-se a definição de TANENBAUN e RENESSE ( 1985 ), por considerar a que melhor faz transparecer os fatores que determinam um Sistema Distribuído.

Em seguida, descrevem-se os aspectos de tolerância a falhas e princípios e conceitos de Sistemas de Computação Tolerante a Falhas.

## II.1 CARACTERIZAÇÃO DE SISTEMAS DISTRIBUÍDOS

Seguindo KIRNER ( 1986 ) e KIRNER e MENDES ( 1988), o advento das redes de comunicação de computadores e do bom domínio de técnicas sobre o compartilhamento dos recursos de "hardware" de comunicação, como canais e processadores de comunicação, deu origem a vários outros tipos de Sistemas de Computação diferentes daqueles já bem conhecidos Sistemas Centralizados, como CP/M, DOS, UNIX, etc.

Assim, um Sistema de Computação pode ser composto de elementos de "hardware" geograficamente distribuídos ( até entre quaisquer países do mundo ), interligados por uma rede de comunicação e, conseqüentemente, os seus elementos de "software" também poderiam estar fisicamente distribuídos. Para tanto, houve um bom desenvolvimento das técnicas de compartilhamento de recursos de computação, como "hardware", programas, banco de dados, etc.

No princípio, houve uma preocupação maior no desenvolvimento de redes para comunicação com computadores que poderiam estar muito distantes entre si, envolvendo satélites para a efetivação da comunicação através de ondas de rádio ou de micro ondas. Porém, o custo da implantação de tal tipo de rede de comunicação era ( e continua sendo ) muito alto. Além disso, devido às perturbações a que esse meio de comunicação fica sujeito, a sua utilização exige um cuidado todo especial ( protocolos sofisticados de comunicação ), aumentando o seu custo e degradando o seu desempenho.

Entretanto, de um modo geral, os benefícios que esses tipos de Sistemas de Computação oferecem, justificam plenamente os investimentos. Apenas para exemplificar, citam-se a RENPAC ( Rede Nacional de Comutação de Pacotes ), a BITNET, por meio das quais se tem à disposição uma vasta quantidade de informações úteis a um custo relativamente baixo ( tipo de rede pública ) e pode-se ter um grande potencial para processamento, desde que se tenha permissão de acessar os recursos computacionais de cada sistema de computação conectado à rede. Um outro exemplo, bastante conhecido por nós é o Sistema de Caixa Eletrônica de bancos.

Pensando em utilizar os benefícios que esse tipo de sistema demonstrava, houve um desenvolvimento, iniciado logo em seguida, das redes locais. Esse tipo de rede de comunicação de computadores, tinha um custo relativamente bem menor do que aquele, devido à restrição da cobertura geográfica que ele permitia. Além disso, ele oferecia confiabilidade e desempenho maiores inerentes, pois, o seu meio poderia ser protegido contra perturbações externas, por meio da utilização de cabos adequados ( por exemplo, cabo blindado ), e assim, o acesso poderia ser bastante simplificado ( padrão IEEE 802, por exemplo ). Normalmente, esse tipo de rede pode interligar processadores que podem estar a um mínimo de distância possível ( podendo ser constituído de barramento ) até a uma distância dentro de um raio de 200 metros, aproximadamente.

Com aquele outro tipo de rede de comunicação, um Sistema de Computação normalmente ( é mais comum ) compartilha seus recursos computacionais de forma explícita. Isto é, o usuário lotado em uma estação ( computador ) da rede, que deseja utilizar os recursos computacionais disponíveis em uma outra estação, deve especificar explicitamente essa intenção, e para tal, deverá conhecer os detalhes operacionais de utilização daqueles recursos. Em palavras simples, pode-se dizer que, um Sistema de Computação desse tipo é composto de vários Sistemas de Computação Centralizados independentes, interligados por uma rede de comunicação, colocando à disposição de seus usuários uma vasta gama de recursos computacionais.

Já com as redes locais, tornou-se viável a construção de Sistema de Computação que compartilha seus recursos computacionais de forma implícita; o acesso aos recursos remotos é feito automaticamente pelo sistema de forma transparente ao usuário. Sistemas desses tipo acabaram, em grande parte, constituindo-se nos Sistemas Distribuídos.

Esse termo Sistema Distribuído, como os termos Processamento de Dados Distribuídos, Processamento Distribuído, Computação Distribuída, têm sido definidos por muitas pessoas de forma diversificada. Por exemplo, é comum definir Sistema Distribuído como aqueles que possuem terminais inteligentes, ou processadores de entrada e saída, ou processadores "front-end". Essa diversidade reside no fato de que cada um procurou colocar em sua definição as características que achou relevante para distinguir esse tipo de Sistema de Computação dos sistemas de arquitetura clássica. A importância de uma definição de consenso é justamente para se evitar o uso não adequado do termo em

questão, como feito logo acima. Por definição de consenso, se quer dizer uma definição que explicite os fatores gerais que realmente determinam as características de um Sistema Distribuído.

Assim, estabeleceu-se que a principal característica é a descentralização do controle. Isto é, dos vários elementos de um sistema que podem ser distribuídos, tais como UCP, memória, dados, programas e controle ( Sistema Operacional ), a distribuição deste último, é o que fundamentalmente caracterizará os Sistemas Distribuídos.

A distribuição do controle está relacionada com o processamento dos programas do sistema, ou seja, com o estado da computação do sistema. Se o progresso da execução dos programas do sistema, em qualquer instante, ficar sob o controle de um único processador, o controle é dito centralizado. Caso contrário, o controle é dito descentralizado; isto é, se o progresso da execução dos programas do sistema passar por mais do que um processador.

Apenas para exemplificar, KIRNER ( 1986 ), cita-se a definição devida a R.H. ECKHOUSE Jr. e outros: um Sistema Distribuído é formado por um conjunto de módulos, compostos pelo menos por um processador-memória, interligados frouxamente por meio de um subsistema de comunicação de topologia arbitrária. Esse "hardware" deve oferecer facilidades de comunicação entre processadores e entre processos, os quais, cooperando sob um controle descentralizado, possibilitam a execução de programas de aplicação.

Em KIRNER ( 1986 ) e KIRNER e MENDES ( 1988 ), estão descritas outras definições. Em todas elas, além da explicitação dessa característica comum, estão outras que, de modo geral, são particularidades, como, por exemplo, na definição devida a A.Z. SPECTOR, é explicitada como característica, a possibilidade de expansão para permitir crescimento incremental. Percebe-se, entretanto, que a não observação dessa característica, não implica em dizer que o sistema não seja distribuído.

Diante da diversidade existente, enfocar-se-á a caracterização de Sistemas Distribuídos segundo TANENBAUN e RENESSE ( 1985 ) que, faz transparecer as seguintes características principais:

a) múltiplos processadores independentes,

b) o ambiente é transparente ao usuário e

c) o Sistema Operacional é integrado e tolerante a falhas.

Por múltiplos processadores independentes entende-se a existência necessária de mais de um elemento processador-memória

interligados frouxamente através de um subsistema de comunicação, de topologia arbitrária, por meio do qual é permitida a comunicação entre processadores.

Pela segunda característica, entende-se que o uso de múltiplos processadores deverá ser transparente ao usuário. Em outras palavras, o usuário deverá ver o sistema como um sistema uniprocessador virtual e não como uma coleção de máquinas.

Quanto à característica de que o Sistema Operacional é integrado, entende-se que ele deve ser construído como um único "software", responsável pelo gerenciamento, de forma descentralizada, de todos os recursos de "hardware" que estão à sua disposição ( o subsistema de comunicação, os processadores interligados ao subsistema, dispositivos periféricos, etc ).

Por último, quanto ao aspecto tolerância a falhas, entende-se que os Sistemas Operacionais Distribuídos podem ser mais confiáveis do que os Sistemas Centralizados, principalmente, diante da descentralização do controle e diante da redundância de recursos de "hardware" inerente.

## II.2 TOLERÂNCIA A FALHAS

Segundo SHRIVASTAVA ( 1985 ), na introdução, a pesquisa para investigar o projeto e construção de Sistemas de Computação Confiável, foi iniciado por B. Randell da Universidade de Newcastle upon Tyne em 1972.

A partir dessa data, o interesse sobre esse tópico da Ciência da Computação cresceu muito. Entretanto, a investigação, com a pesquisa e desenvolvimento, continuava ainda um tanto localizada. Pois, a grande maioria dos artigos referentes a esse tópico era proveniente do grupo de pesquisadores da citada Universidade. Isto é uma suposição, uma vez que podiam existir vários outros pesquisadores, mas por proibição de seus financiadores ou patrões, nada era divulgado; ou porque a pesquisa fosse considerada segredo de Estado ou segredo Industrial, como é o caso de sistema de computação de bordo de espaçonaves.

O entrave maior talvez não fosse pela pouca ( relativa ) difusão de pesquisas sobre esse tópico, mas sim pelo alto custo dispendido para se obter alta confiabilidade. A grande maioria dos sistemas comerciais eram ( e ainda, de certo modo, continuam sendo ) considerados não críticos, isto é, uma falha no sistema implica em uma perda não muito

significativa ( do ponto de vista dos produtores ). Assim, colocava-se o mínimo de confiabilidade, o suficiente para convencer o cliente de que seu produto era muito bom. Na realidade, a preocupação maior era quanto a segurança que o sistema oferecia aos seus usuários. Apesar dos termos de segurança e confiabilidade não serem sinônimos, existem muitas coisas em comum ( veja LINDEN, 1976 ). Assim, tais investigações acabaram sendo particularizadas para determinadas aplicações ( e daí, talvez o não despertar do interesse da maioria dos pesquisadores em Ciência da Computação em relação ao tópico Confiabilidade de Sistemas de Computação ). Como exemplo tem-se os sistemas de computação de bordo de aviões, sistema de computação de satélite, etc; nos quais uma falha poderia causar a perda de vidas humanas ou implicaria em um custo de reparação muito elevado, como é o caso de sistemas acima citados. A confiabilidade desses tipos de sistemas era obtida submetendo o "software" a uma validação ( testes e provas de verificação ) e pela colocação de redundância no "hardware". O custo de tudo isso justificava plenamente a finalidade. Entretanto, obtinha-se dessa forma um sistema não livre de falhas, mas sim, um sistema altamente confiável. Sabia-se que circunstâncias imprevistas ( muito remotas ), poderiam aparecer, como por exemplo, desgaste simultâneo no "hardware" principal e nos redundantes. Além disso, a bateria de testes a que o "software" era submetido no processo de validação, era um conjunto de situações previsíveis e desejáveis, e previsíveis e não desejáveis ( as quais teriam que ser contornadas ). Havia a possibilidade de que alguma situação não desejável pudesse não ser prevista, a qual seria constatada somente quando o sistema estivesse em operação real. Naturalmente, para sistema de "software" de certa simplicidade, no qual se pode prever sem muita dificuldade todas as situações possíveis pelas quais ele pode passar, pode-se afirmar, após a sua validação, que tal sistema está correto com relação à sua especificação; se esta especifica corretamente as funções do sistema, então, o "software" pode ser admitido como totalmente confiável quanto a sua correção.

O avanço tecnológico em "hardware" e conseqüente queda em seu custo, entretanto, permitiram a construção de sistemas de "hardware" cada vez mais complexos, como por exemplo, Redes de Computadores. O "software" para a sua gerência também evoluiu, tornando-se, de modo geral, sistema de "software" de grande porte, cuja complexidade era o

reflexo pela busca da utilização eficiente do "hardware" para a obtenção de máximo desempenho, item muito importante em computação.

Quanto a confiabilidade, esse avanço tecnológico permitiu que o custo da redundância em "hardware", se fosse necessária, diminuísse. Entretanto, o aumento da complexidade do "software" aumentou a incerteza de que ele, mesmo após minuciosa verificação, pudesse ser validado e declarado absolutamente perfeito. A ocorrência de evento não especificado no "software" ( por esquecimento ou não previsão, por exemplo ), aumentou muito a ponto de levar o sistema ao colapso com perdas de trabalho útil, constantemente. Disso, surgiu o interesse em se procurar também, mecanismos de "software" que diminuíssem a frequência de colapsos de sistemas de computação decorrentes de faltas desse tipo, melhorando assim a sua confiabilidade ( HORNING et alii, 1974 ).

Diante desse panorama, é melhor considerar sistemas de computação tolerantes a falhas em vez de sistemas de computação perfeitos, para caracterizar bem o sentido de confiabilidade em sistemas de computação. Em outras palavras, sistemas de computação tolerantes a falhas, são sistemas que, mesmo com a presença de faltas, continuam funcionando ( possivelmente de forma degradada ), mas produzindo resultados confiáveis .

Assim, um sistema de computação que leva em consideração a sua confiabilidade, pode ser classificado como ilustrado pela figura II.1 ( ALFORD et alii, 1985 ).

O sistema perfeito é aquele cujo "hardware" e "software" são perfeitos que diante do exposto, é mais adequado referí-lo como sistema intolerante a falhas. O sistema tolerante a falhas é obtido por meio de redundância de "software" e/ou de "hardware". Essa redundância pode ser estática ou com reconfiguração dinâmica. A redundância estática é geralmente usada em sistema que tem configuração fixa; os seus recursos computacionais são alocados de forma bem determinada e fixa, quando da configuração inicial do sistema. A tolerância a falhas é feita em recurso computacional que permite que ele se recupere a tempo para executar a tarefa pedida. Caso não seja possível, esse recurso entra em colapso, que por sua vez pode provocar o colapso do sistema todo. Na redundância com reconfiguração dinâmica, os recursos que entrarem em colapso, e permanecerem assim, por não ter sido possível sua recuperação em tempo de execução, são automaticamente excluídos do sistema por meio da reconfiguração, para que este continue o seu

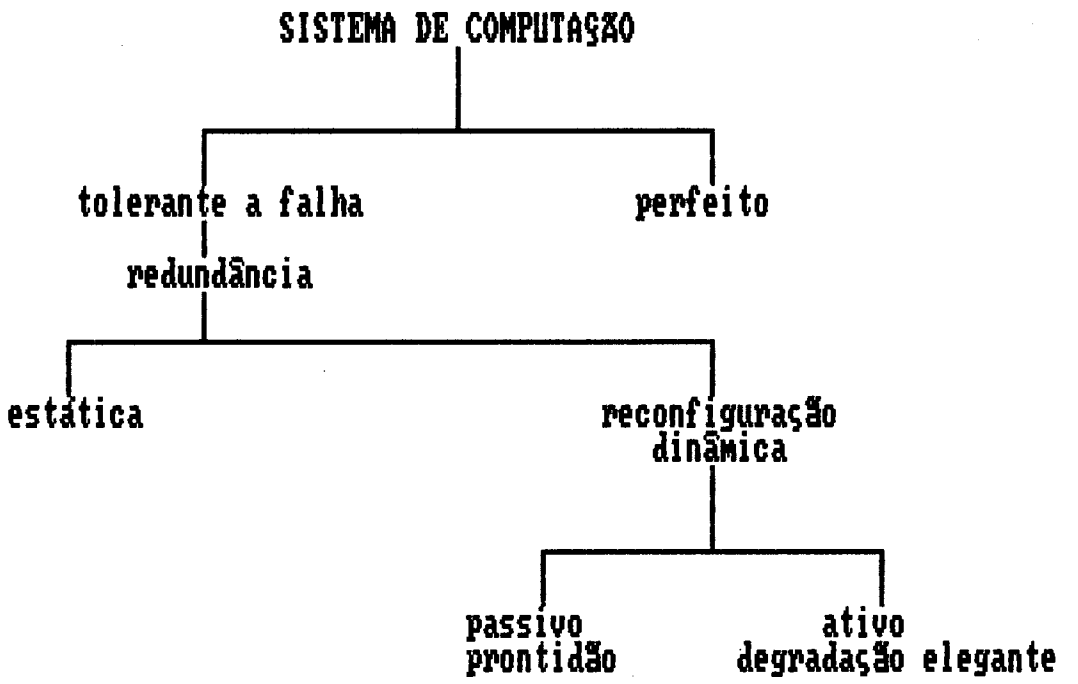



Figura II.1 : confiabilidade de sistema

processamento produzindo trabalho útil.

Assim, dependendo do recurso que entrou em colapso, o sistema continua com o seu processamento na forma degradada ou não. A forma não degradada é possível se existirem recursos de reserva ( "step" ) como, por exemplo, processadores de reserva . E a forma degradada, quando existem recursos equivalentes ( porém, nenhum está inativo na forma de recurso de reserva ) e que seja possível transferir as tarefas de um para um outro sem maiores problemas. O poder de processamento é que diminui devido a diminuição dos recursos disponíveis. No primeiro caso, com recursos de reserva, a tolerância a falha é denominada de tolerância a falha com reconfiguração dinâmica passiva. No segundo, é denominada de tolerância a falhas com reconfiguração dinâmica ativa. Este último é o tipo de tolerância a falhas que atualmente se está tendo grande atenção, sendo aplicável em Sistemas Multiprocessadores e principalmente em Sistemas Operacionais Distribuidos.

## II.2.1 CONFIABILIDADE, PRINCÍPIOS E CONCEITOS DE SISTEMAS DE COMPUTAÇÃO TOLERANTES A FALTAS

Das várias definições de confiabilidade de sistemas sugeridas pelos pesquisadores da área, transcrevem-se duas que julgam-se serem as mais expressivas. Elas são:

*" A confiabilidade de um sistema é uma medida ( métrica ) do sucesso do sistema, obedecendo alguma especificação autoritária de seu comportamento. " - RANDELL ( 1979 ).*

*" Confiabilidade de "software" é a probabilidade de se ter sucesso na execução de um programa de acordo com as especificações, por um dado período de tempo. " - SHOOMAN ( 1979 ).*

A primeira, devida a RANDELL ( 1979 ), procura expressar o significado de confiabilidade de modo geral, enquanto que a segunda, sugerida por SHOOMAN (1979 ), explicita o significado de confiabilidade de "software". Essa é a única diferença, aliás, importante por deixar bem claro o significado de confiabilidade, que é uma probabilidade e não uma certeza absoluta, qualquer que seja o sistema ( o conjunto "hardware" e "software", ou só o "hardware" ou só o "software" ). Uma outra preocupação que se vê em ambas as definições, é quanto ao termo especificação, visando deixar bem explícito a relação íntima entre confiabilidade e especificação.

A especificação de um sistema é um tópico grande e complexo. É uma das primeiras tarefas do processo de desenvolvimento de "software", que antecede a fase da implementação. Os métodos de especificação formal procuram descrever o comportamento externo do programa ou componente de programa, sem descrever ou restringir sua implementação interna. A descrição do comportamento externo do programa ou componente incluirão ( MELLIAR-SMITH e RANDELL, 1977 ):

- que operação ele pode ser solicitado a fazer;
- que informação que deve ser dado a ele;
- que resultados serão obtidos, incluindo indicação de erros; e
- os efeitos de operações a priori sobre as operações subseqüentes.

Mencionou-se especificação formal, pois uma das funções da especificação é evitar que seja permitida mais do que uma interpretação, o que pode ocorrer em uma informal ( a qual

normalmente é a descrição do programa ou do componente de programa através da linguagem natural ).

No presente trabalho, se está preocupado especificamente com os mecanismos, princípios e conceitos envolvidos para a construção de "software" tolerante a faltas. Portanto, maiores detalhes daquele tópico, especificação de sistemas, fogem ao presente objetivo. Todavia, o que foi descrito sobre ela parece o suficiente para mostrar a relação e a sua importância para a compreensão dos problemas de confiabilidade e, assim, de tolerância a faltas.

Um ponto interessante sobre tolerância a faltas é quanto a diferenciação entre os termos mais comumentes utilizados: falhas, colapsos, faltas e erros . Existem diferenças semânticas nesses termos que merecem serem aqui esclarecidas para evitar confusões em seus empregos no presente trabalho ( vários artigos anotados na bibliografia as fazem, por exemplo, ANDERSON e LEE ( 1982 ), RANDELL ( 1980 ), KIRNER ( 1986 ) e KIRNER e MENDES ( 1988 ), STONE ( 1989 ) e NELSON ( 1990 ). Assim:

- uma **falha** é o evento que levou o comportamento do sistema a se desviar daquele requerido pela sua especificação;
- Um **colapso** do sistema é a falência do sistema;
- Uma **falta** no sistema, significa a ocorrência de um defeito físico ( dispositivo físico ou simplesmente um "chip" ) ou de algoritmo ( projeto lógico - falta residual de projeto ( HORNING et alii, 1974 ) );
- Um **erro**, significa a existência de um estado defeituoso ( incorreto ) no sistema.

Esses eventos não surgem isoladamente. Normalmente um erro é um item de informação que quando processado por algoritmos normais, produzirá uma falha. Entretanto erros nem sempre produzem falhas; eles podem, por exemplo, serem expurgados por algoritmos de recuperação. Uma falta pode gerar um erro. E a falha de um componente pode gerar um erro que aparecerá como uma falta em qualquer lugar do sistema. Desta forma, portanto, pode-se, resumidamente, dizer que faltas levarão a erros que, se não forem expurgados por algoritmos de recuperação, levarão o sistema ao colapso.

A seguir faz-se uma breve explanação sobre os princípios e conceitos propostos por alguns pesquisadores para a construção de sistemas de computação ( "hardware" e "software" ) tolerantes a faltas.

DENNING ( 1976 ), desenvolve quatro princípios relacionados, baseados no princípio fundamental de Sistema Fechado, denominados de princípios de confinamento de erros. Os quatros princípios que ele acredita que podem guiar a construção de Sistemas Operacionais Tolerantes a Faltas, são:

- **isolação de processo:** é o princípio de que cada processo não deverá ter mais capacidade além daquela necessária para a realização da tarefa.
- **controle de recurso:** ( implementa a atribuição de unidades de recursos físicos a objetos computacionais; por exemplo, processadores a processos, quadro de páginas a segmentos ) é o princípio de que, quando uma unidade for adquirida por um novo objeto através de preempção, o estado da unidade deverá ser salvo e a unidade colocada no estado nulo; e quando for reatribuida, a unidade deverá ser colocada no estado que estava quando da última preempção, quando estava ocupada pelo objeto que agora a deseja novamente. Resumindo, nenhuma unidade deverá conter vestígios de objetos distintos.
- **verificação da decisão:** princípio que especifica que cada decisão deverá ser computável no mínimo de dois modos diferentes. Uma discrepância indica um erro.
- **recuperação de erro:** princípio que diz para procurar reparar danos; se isto não é possível, procurar reconfigurar o sistema, removendo do serviço quaisquer unidades de recursos defeituosos e colocando as unidades de recursos remanescentes e processos, em estados consistentes.

O primeiro princípio impõe a regra de que os processos só podem interagir através de caminhos pré-definidos, impedindo que ocorram formas não esperadas de interferências. Implica também que a informação que descreve as capacidades de um processo deve ser protegida contra alterações.

O segundo princípio especifica a habilidade de verificar quais comandos são apropriados para se atribuir e liberar: uma unidade poderá ser atribuida a um objeto, somente quando estiver livre, e liberada somente quando estiver ocupada.

O terceiro princípio especifica a permissão de que os processos cooperantes aprovem as mensagens trocadas entre si, por exemplo.

O último princípio é decorrência direta dos três primeiros que permite que os erros sejam detectados e localizados antes que eles

possam se propagar. A recuperação de erro em tais sistemas pode realmente procurar corrigir os erros, em vez de simplesmente abrandar seus efeitos.

A principal conclusão que DENNING ( 1976 ) chegou foi que, Sistemas Operacionais são não confiáveis devido às inadequações de assistência de "hardware" para a detecção e confinamento de erros em "software". Devido a busca tradicional da flexibilidade, uma pesada sobrecarga é associada com a imposição da estrutura lógica de Sistemas Operacionais em "hardwares" amorfos. Conseqüentemente, a estrutura adicional, para a verificação de erros e confinamento, não podem ser suportados por "hardware" tradicionais. Sistemas Operacionais altamente confiáveis serão obtidos, somente quando o núcleo do Sistema Operacional for construído com integração do "hardware" e "software". Isto significa que a máquina abstrata, consistindo do núcleo, é especificada primeiro, e somente então são projetados os requisitos de "hardware" ( ou "firmware" ), deixando que as considerações de custo e eficiência determinem quais porções da máquina abstrata serão implementadas em "hardware". Para tal, DENNING ( 1976 ) propõe uma máquina baseada em capacidade.

LINDEN ( 1976 ) propõe dois conceitos para estruturação de sistemas que suportam segurança e que também suportam confiabilidade. Entretanto, sua preocupação maior é quanto a segurança e procura, através dela, mostrar a relação com a confiabilidade do sistema assim estruturado. Os conceitos são:

- **pequenos domínios de proteção:** permite que cada sub-unidade ou módulo de um programa seja executado num ambiente restrito, que pode evitar ações indesejáveis ou não previstas para aquele módulo.
- **Objetos de tipos estendidos:** provê um veículo para a abstração de dados, permitindo que objetos de novos tipos sejam manipulados em termos de operações que sejam naturais àqueles objetos.

LINDEN ( 1976 ) também considera uma abordagem voltada a implementações eficientes, baseada no conceito de capacidade incorporada na estrutura de endereçamento do computador. Conclui que o endereçamento baseado em capacidade parece ser um meio prático para suportar futuros requisitos para a segurança e confiabilidade em "software", sem sacrificar requisitos de desempenho, flexibilidade e compartilhamento.

Note que o conceito de pequenos domínios de proteção é equivalente ao princípio de isolação de processo, e o conceito de tipos de dados estendidos engloba o princípio de controle de recursos.

Problemas de tolerância a faltas em sistemas de computação são descritos de forma mais detalhada, nos artigos de SIEWIOREK ( 1984, 1990 ). Ele define que um sistema redundante ( tolerante a falta ) pode passar no máximo por até dez estágios em resposta a ocorrência de falhas, a saber:

- **confinamento da falta:** quando ocorrem faltas, é desejável limitar o escopo de seus efeitos. O confinamento da falta é obtido, limitando a propagação de seus efeitos a uma área do sistema, prevenindo, desta forma, a contaminação das outras áreas.
- **detecção da falta:** a maioria das falhas eventualmente resultam de faltas lógicas. Estão disponíveis muitas técnicas para detectar faltas, tais como paridade, verificação de consistência, e violação de protocolo. Infelizmente, essas técnicas não são perfeitas, e pode ter passado um período arbitrário de tempo antes que ocorra a detecção. Esse tempo é chamado de "latência da falta". As técnicas de detecção de faltas são de duas grandes classes: detecção "off-line" e detecção "on-line". Com a detecção "off-line", o dispositivo não é capaz de realizar trabalho útil enquanto estiver sob teste. Programas de diagnósticos, por exemplo, rodam no modo isolado, mesmo se executado sobre dispositivos ociosos ou multiplexados com as operações de "software". Assim, a detecção "off-line" assegura a integridade antes e possivelmente em intervalos durante a operação, mas não durante o tempo todo da operação. A detecção "on-line", por outro lado, provê uma capacidade de detecção em tempo real, para isto ela é realizada concorrentemente com o trabalho útil. As técnicas "on-line" incluem detecção e duplicação de paridade.
- **mascaramento da falta:** as técnicas para mascaramento de faltas escondem os efeitos das falhas. Num sentido, a informação redundante alivia o peso da informação incorreta. Em sua forma pura, o mascaramento não provê nenhuma detecção. Entretanto, muitas técnicas para mascaramento de faltas podem ser estendidas para prover também detecção "on-line". Caso contrário, são necessárias técnicas "off-line" para descobrir falhas. A maioria votante é um exemplo de mascaramento de faltas.

- **tentar novamente:** em muitos casos, uma segunda tentativa de uma operação pode ter sucesso. Isto é particularmente verdadeiro para faltas transientes, que não causam nenhum dano físico.
- **diagnóstico:** se a técnica de detecção de falta não provê informação sobre a localização da falha e/ou propriedade, pode ser necessário um passo de diagnóstico.
- **reconfiguração:** se uma falta é detectada e um colapso permanente localizado, o sistema pode ser capaz de configurar seus componentes para substituir o componente estragado ou isolá-lo do resto do sistema. O componente pode ser substituído por reservas. Alternativamente, ele pode ser simplesmente desligado e a capacidade do sistema ficar degradado: este processo é chamado de "degradação elegante".
- **recuperação:** após a detecção e ( se necessário ) a reconfiguração, os efeitos dos erros devem ser eliminados. Normalmente, a operação do sistema é retrocedida a algum ponto em seu processamento que preceda a detecção da falha, e a operação recomeça deste ponto. Esta forma de recuperação, freqüentemente chamada de rolar para traz ( "rollback" ), geralmente requer estratégias que usam cópias de arquivos, pontos de verificação, e diário de ocorrência. Na recuperação, a latência do erro torna-se uma questão importante, porque deve-se recuar o suficiente para evitar os efeitos de erros não detectados que ocorreram antes daquele detectado.
- **reinício:** a recuperação pode não ser possível, se o erro estragou muita informação, ou se o sistema não foi projetado para ser recuperável. O reinício quente, isto é, a retomada de todas as operações a partir do ponto da detecção da falta, é possível somente se não ocorreu nenhum dano físico. Um reinício morno implica que somente alguns dos processos podem ser retomados sem perdas. Um reinício frio corresponde a uma recarga completa do sistema, sem a sobrevivência de nenhum processo.
- **reparação:** quando um componente é diagnosticado como estragado, ele é substituído. A detecção e a reparação da falta ( substituição dos componentes estragados ) podem ser feitas com o sistema fora de operação ou em operação conforme a figura II.2. No caso fora de operação, o componente estragado deve ser removido para reparo, podendo ou não implicar na desativação do sistema. Na reparação em operação, o sistema isola automaticamente o

componente, passando a funcionar sem ele, que é o caso da redundância com mascaramento ou degradação elegante. Note que nesse caso, o componente estragado pode ser fisicamente substituído ou reparado sem a interrupção da operação do sistema.

- **reintegração:** após a substituição física de um componente, o módulo reparado deve ser reintegrado ao sistema. Para a reparação em operação, a reintegração deve ser realizada sem a paralização da operação do sistema.

SIEWIOREK ( 1984 ), levando em consideração esses estágios em resposta a colapso de sistema, dividiu o espectro das técnicas para tolerância a faltas em três grandes classes, a saber:

- detecção de faltas,
- mascaramento por redundância e
- redundância dinâmica.

A primeira classe se refere a implementação de apenas de técnicas para a detecção de faltas; não é provida nenhuma técnica para mascarar faltas, salvo aquelas que são transientes ( como, por exemplo, "driver" de disquete inativo ). Essa classe de técnicas é muito utilizada emsistemas de microcomputadores e minicomputadores. A reparação das faltas é feita com o sistema desativado e, geralmente, é também utilizado um programa de diagnóstico para a detecção das faltas

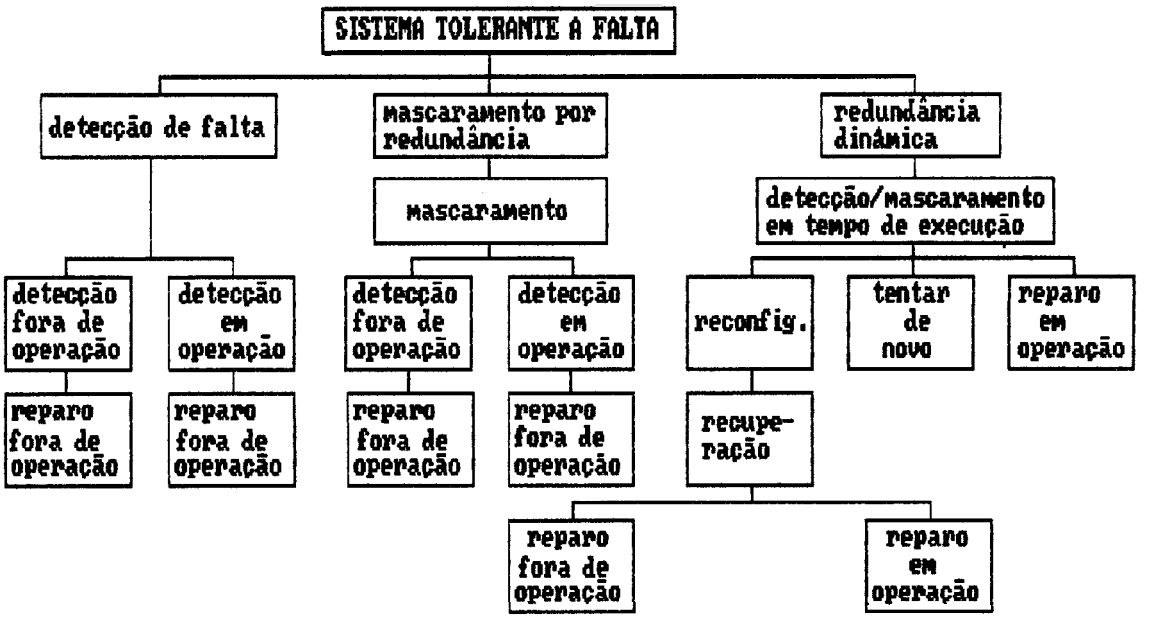



Figura II.2: Taxonomia das estratégias de sistemas tolerante a faltas ( SIEWIOREK, 1984 ).

não registradas pelos mecanismos de detecção em tempo de execução. As outras duas classes, já foram citadas, as quais denominaram-se de tolerância a faltas por redundância estática e por redundância dinâmica, respectivamente.

Um novo termo sugerido por Laprie e referenciado por STONE ( 1989 ), é credibilidade ( "dependability" ),

*"Credibilidade de Sistemas de Computador é a qualidade dos serviços deliberados tais que a confiança pode ser justificavelmente colocada sobre eles",*

introduzido como uma nova medida de desempenho de sistema, para mostrar de forma bem clara as diferenças entre a confiabilidade de modo geral e aquela medida matematicamente. A credibilidade descreve um atributo geral de um sistema que inclui medidas como confiabilidade, segurança e disponibilidade. Essa terminologia pode ser agrupadas em três classes de atributos, a saber:

a) **Danos:** circunstâncias onde a confiança não pode ser mais colocada sobre o serviço;
b) **meios:** os métodos e ferramentas que permitam que um serviço possa ser justificavelmente dito ser de confiança;
c) **medidas:** a importância da qualidade do serviço seguinte aos danos e a avaliação dos meios que se opõem aos danos.

Esquematicamente, a terminologia sugerida por Laprie é a ilustrada pela figura II.3.

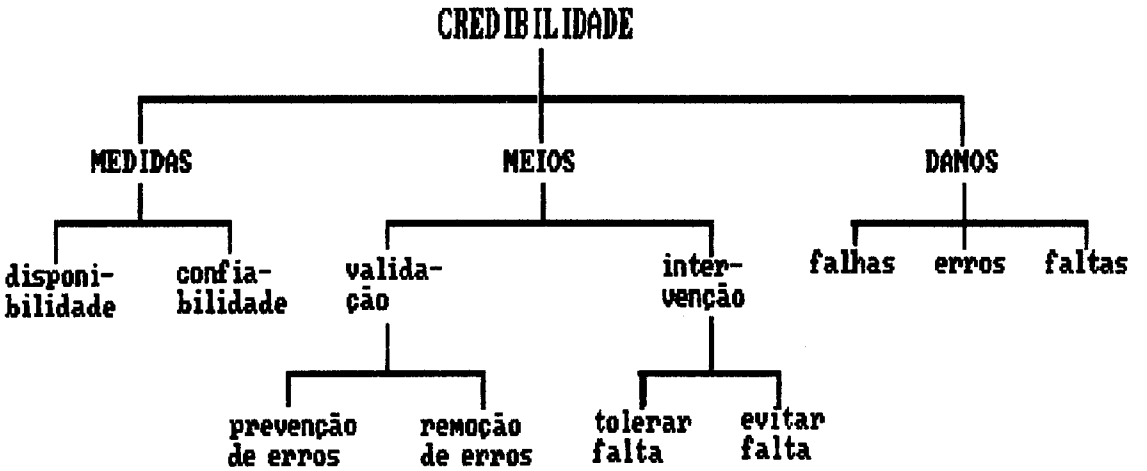



Figura II.3: Árvore de terminologia de Laprie

# CAPÍTULO III

## ESTRUTURAÇÃO DE SISTEMAS DISTRIBUíDOS TOLERANTES A FALHAS

Como já é bem conhecido, o "software" de sistemas, principalmente os Sistemas Operacionais, sofreram evoluções com o passar do tempo diante das evoluções tecnológicas do "hardware" e dos princípios, métodos, técnicas e mecanismos do próprio "software". Quanto ao "hardware", o avanço tecnológico permitiu a construção de redes de computadores, redes locais de computadores, supercomputadores, etc. Como não poderia deixar de ser, o "software" precisou acompanhar essa evolução, pois é ele o responsável pela moldagem desses equipamentos aos objetivos por ele definidos. Um exemplo clássico, o conhecido Sistema Operacional de Multiprogramação, molda um computador monoprocessador para permitir que mais do que um usuário utilize o equipamento simultaneamente. Um outro exemplo, uma rede local de computadores pode ser moldado para se constituir em um Sistema Operacional Distribuído ( único para todos os computadores da rede ), ou em um Sistema Operacional de Rede de Computadores, adaptando os Sistemas Operacionais existentes em cada computador da rede para permitir que um terminal seja conectado ao computador que

se deseja utilizar. Outros exemplos, Sistemas Operacionais para máquina "Data Flow", para computadores com "array processors" - supercomputadores, etc.

Obviamente, além da moldagem de um equipamento computacional, um Sistema Operacional deve otimizar a utilização dos recursos físicos disponíveis e, principalmente, oferecer uma alta confiabilidade ( capítulo II ). A busca por esses requisitos foram tornando o desenvolvimento desse tipo de "software" cada vez mais complexo.

Diante disso, foram elaborados princípios, métodos, etc., que diante do crescente desenvolvimento de novas arquiteturas de máquinas, como as acima citadas, visam facilitar e até tornar sistemática a construção de "software" desse tipo. Um princípio muito bem conhecido é a estruturação de sistemas em uma hierarquia de camadas, cada uma implementando um nível de abstração, caracterizado pelos objetos colocados à disposição, juntamente com as operações para manipulá-los. A camada de "software" mais interna, que trata dos detalhes da máquina e que dá o suporte para as demais camadas, é denominado de "nucleous", "kernel" ou "monitor" ( HANSEN , 1977, 1982; LISTER, 1983; PETERSON e SILBERSCHATZ, 1983,1985; DEITEL, 1984; HOLT, 1983, etc ). A figura III.1 ilustra a estruturação de um sistema de computação dessa forma.

Neste trabalho o interesse repousa nos princípios, métodos, etc., que objetivam combater a complexidade do desenvolvimento de Sistemas Operacionais Distribuídos que apresentem um alto grau de confiabilidade ( Sistemas Distribuídos Tolerantes a Falhas ). Uma complexidade é decorrente da própria arquitetura de "hardware" que, de um modo geral, apresenta muitas interligações físicas de

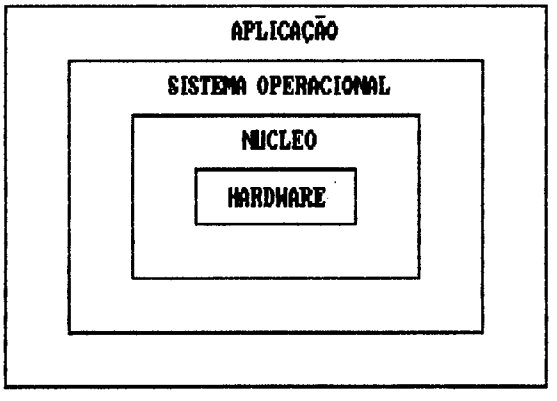



Figura III.1 Estruturação de um sistema por camadas

componentes não totalmente confiáveis e, portanto, a ocorrência de um defeito em um deles pode levar todo o sistema ao colapso, se não for devidamente contornado; às vezes, é necessário que o "hardware" ofereça infra-estrutura para tal. Uma outra complexidade é proveniente do desenvolvimento do projeto. Devido ao seu porte em termos da quantidade de tarefas que deve ser executa como funções do Sistema Operacional e da quantidade de mecanismos diferentes necessários para tratar todas as possíveis falhas passíveis de ocorrerem, o próprio Sistema Operacional pode acabar contendo erros.

Assim, descreve-se a seguir, os conceitos e técnicas para a estruturação de Sistemas Distribuídos Tolerantes a Falhas, iniciando-se pela definição de "software" distribuído.

## III.1 "SOFTWARE" DISTRIBUÍDO

Dos quatro elementos de um sistema: o "hardware", os dados, os programas e o controle, que podem estar distribuídos, os três últimos se referem ao "software".

A distribuição dos dados se constitui em uma grande área que envolve sistemas de arquivos distribuídos e sistemas de banco de dados distribuídos. Tal tópico não será abordado neste trabalho, apesar de sua importância, principalmente, no que se refere a sistema de arquivo, parte integrante de qualquer Sistema Operacional.

Para a compreensão da distribuição de programas ( algoritmos ), é necessário entender o seguinte ( KIRNER, 1986 e KIRNER e MENDES, 1988 ):

- existem programas seqüenciais, cujas instruções são executadas uma por vez e, programas paralelos ou concorrentes, cujas instruções, em conjuntos bem definidos, podem ser executados ao mesmo tempo;
- um programa é dito centralizado, se o processador ( ambiente ) puder executar potencialmente qualquer uma de suas instruções;
- um programa é dito distribuído, se estiver espalhado por vários ambientes, e se alguns processadores ( ambientes ) não puderem executar alguns passos do programa, pelo fato deles pertencerem a outros ambientes;

- qualquer programa seqüencial ou paralelo pode ser um programa distribuído.

Pode-se dizer, assim, que qualquer programa que for executado em um ambiente, onde seus processadores possam executar qualquer instrução desse programa, será um programa centralizado. Por exemplo, sistema multiprocessador com uma única memória compartilhada.

Um programa será distribuído, se ele puder ser estruturado em várias partes, cada uma agrupando um conjunto de instruções que executam uma determinada parte bem definida da tarefa do programa. Esses componentes só podem ser executados pelos processadores a eles associados e, principalmente, se comunicam, para atingir os objetivos do programa, somente por meio de um subsistema de comunicação, através da troca de informações. Os componentes do programa separados pelo subsistema de comunicação são conhecidos como remotos uns aos outros. Assim, um programa seqüencial formado pela sua parte principal e uma subrotina, localizados de forma a serem remotos um ao outro, é um programa distribuído.

A distribuição do controle está relacionada com o processamento do programa, ou seja, com o estado da computação, significando a contrapartida dinâmica do programa. Se o progresso completo da execução de um programa, em qualquer instante, ficar sob o controle de um único processador, o controle é dito centralizado, independentemente se o programa é ou não distribuído.

Agora, se o progresso da execução do programa passar por mais do que um processador, durante o processamento, o controle será distribuído. Exemplos, chamada remota de procedimento, onde o controle é temporariamente transferido para o processador capaz de executar tal procedimento e em seguida retornado ao processador que executou a chamada. O processamento em cadeia com transferência de resultados intermediários, onde o controle é definitivamente transferido de um processador para um outro.

## III.2 ESTRUTURAÇÃO EM CAMADAS

Um princípio muito bem conhecido e defendido para a construção de sistemas de "software" complexos, é estruturar um sistema em uma hierarquia de camadas ( interfaces ). Cada interface implementando um nível de abstração, caracterizado pelos seus objetos colocados à

disposição, juntamente com as operações para manipulá-los. Sistemas assim estruturados, ANDERSON et alii ( 1978 ) e ANDERSON e LEE ( 1981 ), denominaram de sistemas multiníveis, que para melhor examinarem as questões sobre a provisão de recuperabilidade ( tolerância a falhas ), classificaram-nos em:

1) Sistemas multiníveis interpretativos e
2) Sistemas multiníveis interpretativos estendidos.

Suas descrições estão a seguir.

## SISTEMA MULTINÍVEL INTERPRETATIVO

Um sistema multinível é interpretativo quando cada interface implementa todos recursos que os programas que ela executa necessitam.

"Melhor esclarecendo, na figura III.2, cada interface $N_i$ é implementado por meio de um programa $I_i$, que é executado pela interface $N_{i-1}$. Cada interação com a interface $N_i$, que correspondente à execução de uma operação do programa $I_{i+1}$, é, de fato, diretamente suportado pelo programa $I_i$. Qualquer objeto abstrato disponível em $N_i$, tem uma representação concreta como um conjunto de objetos em $N_{i-1}$, que são gerenciados e mantidos por $I_i$. Tal programa $I_i$, é referido como um interpretador para $N_i$."

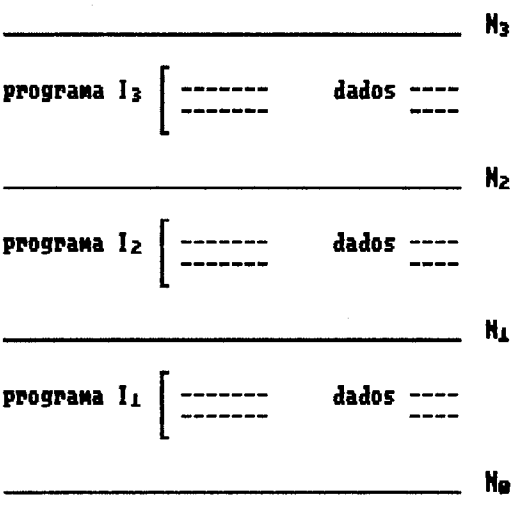



Figura III.2 Sistema Multinível Interpretativo.

## SISTEMA MULTINÍVEL INTERPRETATIVO ESTENDIDO

Nessa categoria se encaixam os sistemas multiníveis onde cada interface implementa alguns dos recursos que os programas que ela executa necessitam e os outros recursos são suportados pelas interfaces inferiores. Isto é, cada interface é uma extensão da anterior.

Segue o esquema ilustrativo da figura III.3 e as explicações mais detalhadas a respeito desse tipo de sistemas multiníveis.

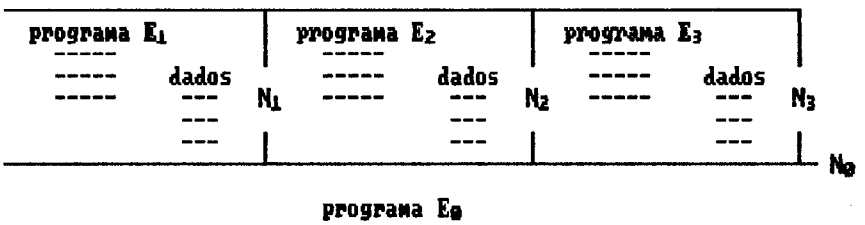



Figura III.3 Sistema Multinível Interpretativo Estendido

"A interface $N_0$ é suportado pelo interpretador $E_0$, que provê facilidades de extensão. Cada interface $N_i$ ( i = 1,2,3 ), é construída como uma extensão da interface $N_{i-1}$, implementada por meio do programa $E_i$, que é executado sobre a interface $N_{i-1}$. O programa $E_i$ é referido como um interpretador extensão. Cada interação com a interface $N_i$, é primeiro examinado pelo interpretador subjacente $E_0$, que determina se aquelas interações são diretamente suportadas por ele, se não, quais das extensões disponíveis ( $E_j$, j <= i ) suportam. Assim, as interações de um programa podem ser suportadas por quaisquer extensões de nível inferior e pelo interpretador básico ( $E_0$ ). Qualquer objeto abstrato disponível em $N_i$, outros que não aqueles diretamente suportados por $E_0$, tem a representação concreta como um conjunto de objetos em uma das interfaces $N_j$ ( j < i ). Esse conjunto de objetos é gerenciado e mantido por $E_{j+1}$."

## OBSERVAÇÕES SOBRE OS DOIS TIPOS DE SISTEMAS MULTINÍVEIS

a) A distinção esquemática é para enfatizar bem a diferença entre os dois tipos de sistemas multiníveis quanto a: questões das propriedades de comportamento entre uma

interface e aquelas abaixo dela; e os diferentes mecanismos usados para implementar as interfaces dos dois sistemas. No sistema multinível interpretativo estendido, o interpretador $E_0$ é responsável pelos programas executados sobre todas as interfaces $N_i$ ( com assistência das extensões quando necessárias ), enquanto que no outro sistema, cada interpretador tem somente a responsabilidade pela interface que ele cria. Para os usuários comuns do sistema, isso deve ser, preferencialmente, transparente ( ocultados ).

b) a estruturação de sistema por meio de extensões de interpretadores exige um esforço de implementação menor que por meio de interpretadores auto suficientes.

c) naturalmente ambas as técnicas podem ser utilizadas para a construção de um sistema estruturada no esquema de multinível.

## III.3 ESTRUTURAÇÃO LÓGICA

A estruturação lógica, que corresponde à estruturação dos elementos componentes do "software", segue o modelo de processos ou o modelo de objetos ( KIRNER, 1986; KIRNER e MENDES, 1988; TANENBAUMM e MULLENDER, 1981; BACON, 1981 e FORTIER, 1986 ).

### A) MODELO DE PROCESSOS

O modelo de processos, como o próprio nome diz, se baseia na utilização do conceito de processo para a estruturação de todo o sistema. Todos os elementos do sistema são encapsulados por processos; tanto os elementos passivos, representados pelos recursos e suas operações, devidamente confinados em gerenciadores de recursos, como os elementos ativos - os programas de controle das atividades do sistema e os programas dos usuários. Os processos que encapsulam elementos ativos são, de um modo geral, denominados de processos clientes e os que encapsulam os elementos passivos, de processos servidores. Essa denominação procura, justamente, expressar as funções desses dois tipos de processos em um sistema. Os processos clientes, através da programação da seqüência dos serviços prestados pelos

processos servidores, definem as suas tarefas dentro do contexto dos objetivos do sistema. Para isso, há a necessidade da existência de um meio de comunicação entre os processos clientes e os processos servidores, como também entre si. Além disso, como os programas de controle das atividades do sistema devem ser elaborados de forma a otimizarem a utilização dos recursos do sistema, os processos clientes se competem para os seus usos, necessitando, assim, de mecanismos de sincronização.

Dessa forma, nesse modelo, os processos clientes se comunicam com os processos servidores ( solicitação de serviços ) e se competem entre si ( concorrência ), por meio de chamada remota de procedimentos ou por meio de troca explícita de mensagens. Tais meios de comunicação e sincronização entre processos, serão tratados no capítulo 4.

## B) MODELO DE OBJETOS

O modelo de objetos baseia-se na utilização do conceito de objetos para a estruturação de todos os elementos do sistema. De forma análoga ao modelo anterior, os objetos se dividem em dois grupos, a saber:

- aqueles que encapsulam processos, que depois de iniciados, constituem a parte ativa do sistema, sendo chamados simplesmente de processos.
- aqueles que encapsulam recursos e facilidades, cuja função resume-se em atender passivamente as requisições provenientes dos processos, sendo chamados simplesmente de objetos.

O conceito central é que um sistema é visto como uma coleção de objetos, existindo em um único nível aparente de espaço de endereçamento. Cada objeto pode ser considerado como um espaço de endereço identificável. Um processo pode endereçar um conjunto desse tipo de espaço. Assim, aquela tradicional relação entre processo e espaço de endereço, em que um espaço contém um processo em execução, é invertida e um processo passa a endereçar um conjunto de espaços endereçáveis.

Dessa forma, é função principal do Sistema Operacional, tornar os objetos conhecidos aos processos sobre a base de alguma estrutura de controle de acesso. A terminologia comum utilizada para dizer que um processo tem acesso a um objeto, é a capacidade. Em outras palavras,

um processo consegue acessar um objeto, se e somente se ele possui uma capacidade para tal. Uma capacidade é um nome de um objeto e uma lista dos direitos que o processo tem sobre o objeto. As capacidades que um processo possui são geralmente colocados em um objeto especial, denominado de lista de capacidades. A alteração dinâmica de uma lista de capacidades, como também a alteração de algum direito deve ser feita de maneira bastante segura para não colocar em risco a integridade do sistema.

## III.4 CONCEITOS E TÉCNICAS PARA ESTRUTURAÇÃO DE SISTEMAS DISTRIBUÍDOS TOLERANTES A FALHAS

Sistemas Distribuídos, como visto, provêem novas possibilidades de construção de Sistemas de Computação de alto desempenho. Entretanto, ao mesmo tempo, apresentam problemas de confiabilidade, ou melhor, de tolerância a falhas, não encontrados normalmente em Sistemas Centralizados. Os problemas são decorrentes de:

a) perda de mensagem pelo subsistema de comunicação;
b) rompimento da linha de comunicação do subsistema de comunicação;
c) colapso de nodo de comunicação do subsistema de comunicação;
d) colapso de nodo operador;
e) falhas transientes de máquinas hospedeiras ( nodos operadores, estação de trabalho );
f) falhas de algum componente de um nodo operador, como parte de sua memória ou interface com algum dispositivo periférico, ou etc.; que não leva a máquina ao colapso;
g) colapso de um dispositivo periférico;
h) erros de "software";
i) desativação de nodos operadores;
j) ativação de nodos operadores;

As faltas a), b) e c), poderão surgir devido a inexistência de um meio de comunicação perfeito entre os nodos operadores, componentes do Sistema Distribuído. As faltas de d) a h) são as que podem surgir nos nodos operadores. O evento i), se ocorrer, pode ser considerado como colapso do nodo e, por último, o evento j), aparece ao se permitir

a reativação de nodos operadores que estavam fora de operação, porém, faziam parte do sistema desde o seu início.

Os problemas listados são os passíveis de ocorrerem em Sistemas de Computação Distribuída. Assim, a fim de se evitar que o surgimento de um deles leve o sistema todo ao colapso, geralmente com grandes perdas de trabalho útil, são necessárias medidas de prevenção ( proteção de memória, segurança em sistema de arquivo, etc ), detecção e correção de erros, a serem tomadas; tornando-o um sistema tolerante a falhas ( veja item anterior ). Assim, foram desenvolvidos vários conceitos e técnicas como bloco de recuperação, tratamento de exceções, tratamento de órfãos, mecanismos para migração de processos, mecanismos para recuperação de processos, etc., que estão descritos mais adiante. Note que esses conceitos e técnicas procuram dar soluções aos vários estágios que um Sistema Tolerante a Falhas pode passar em resposta a ocorrência de falhas, conforme definido por SIEWIOREK ( 1984 ).

Preocupados em não tornar mais complexo do que normalmente é um "software" do porte de um Sistema Operacional sem essa característica de tolerância a falhas, é fundamental o esquema utilizado para a estruturação de seus componentes ( por exemplo, o esquema hierárquico - multinível descrito ), como também a estruturação de cada componente ( por exemplo, em módulos ( DeREMER e LEVY, 1976 ). Assim, tais medidas para tolerância a falhas foram elaboradas com essas preocupações, tanto é que muitas delas são propostas como métodos para estruturação de sistemas com essa característica. Um outro fator importante é a existência de um ambiente ( linguagem de alto nível para programação , compilador, carregador/ligador ( "linker" ), depuradores, etc. ) disponível para a sua implementação; a análise de tal assunto foge aos objetivos deste trabalho e será feita apenas a análise sobre as linguagens de alto nível, de forma resumida, no capítulo V.

Como não poderia deixar de ser, os resultados de pesquisas sempre instigam a procura de outros melhores ou de novas opções, analisando outros modelos totalmente distintos daqueles já utilizados para verificar suas validades para esses, ou modificando-os em alguns de seus aspectos ou, simplesmente adaptando-os para casos específicos, etc. Assim, as primeiras propostas validadas ( teoricamente ou na prática ), têm, no mínimo, seus valores incontestáveis de serem os pioneiros. Naturalmente, não há necessidade de sempre se recorrer a eles se existirem outras mais recentes, ou melhor, os que mais se

aproximam dos que se tem interesse em salientar, para fins de esclarecimento.

Assim, serão descritos os conceitos e técnicas, que se julgaram serem os mais adequados, propostos para solucionar os problemas listados.

## III.4.1 BLOCO DE RECUPERAÇÃO

O bloco de recuperação, originalmente proposto por HORNING et alii ( 1974 ), provê um meio de se introduzir tolerância a falhas de "software" do tipo não antecipadas ( falhas residuais de projeto ), que são mascaradas por meio de redundância e, de modo disciplinado, em processos seqüenciais simples.

Dos vários estágios contra os erros, o conceito de bloco de recuperação trata de quatro, são eles:

- detecção da falha ( detecção do erro );
- confinamento da falha ( avaliação dos danos );
- recuperação ( recuperação do erro ) e
- mascaramento da falha ( tratamento da falha ).

O esquema proposto para a implementação de tal conceito, teve como base a estrutura de bloco da linguagem ALGOL, visando justamente a utilização do conceito de programas bem estruturados ( construção de operações - blocos - bem definidas a partir de outras operações - blocos - menores ) e a exigência de um mecanismo associado para a recuperação de erros, que foi denominado de "cache" de recuperação ou recursivo.

Assim, um bloco de recuperação que tem a função de executar uma determinada operação de modo a tolerar falhas ( provavelmente nem todas que possam ocorrer ), é composto de um bloco primário, um teste de aceitação, e zero ou mais blocos alternativos.

O bloco primário corresponde exatamente a função de executar a operação desejada. O teste de aceitação é executado ao término do bloco primário ou de um alternativo, para confirmar se a operação desejada foi executada de forma aceitável. Os blocos alternativos são as redundâncias da operação do bloco primário. Cada um deve executar a operação desejada de um modo diferente do primário e de outros alternativos, ou realizar alguma ação alternativa aceitável pelo programa como um todo.

O mecanismo para recuperação de erros é colocado automaticamente em execução quando o teste de aceitação falha, para apresentar ao bloco alternativo a ser ativado, o mesmo ambiente encontrado pelo bloco primário.

A semântica de um bloco de recuperação é a seguinte:

1) é colocado em execução o bloco primário;
2) é executado o teste de aceitação;
3) se o teste de aceitação é aceito, é encerrada a execução do bloco de recuperação;
4) se o teste de aceitação falha ( é detectado um erro no bloco primário ), é colocado em execução um bloco alternativo, que deve executar a operação desejada de um modo diferente ou realizar alguma ação alternativa aceitável pelo programa como um todo. Após isso, o teste de aceitação é novamente executado para verificar a aceitabilidade dos resultados deste bloco alternativo; e assim segue até que a execução de um bloco alternativo satisfaça o teste de aceitação ou todos os blocos alternativos fracassam, o que corresponde ao fracasso do bloco de recuperação.

Note, dessa forma, que a detecção de erro fica a cargo do teste de aceitação e o erro fica confinado ao bloco executado ( primário ou alternativo ); assim ficam avaliados os danos ( todo o bloco executado ). A recuperação do erro fica a cargo de um mecanismo subjacente ( o "cache" de recuperação ou recursivo ), para restaurar toda a informação danificada pelo erro para a execução de um bloco alternativo. Por último, o tratamento da falha, é feito pelos próprios blocos alternativos.

Para fins ilustrativos, descrevem-se a sintaxe proposta por RANDELL ( 1975 ), para a representação formal da estrutura de bloco de recuperação e os exemplos.

Sintaxe:

<bloco de recuperação> ::= **ensure** <teste de aceitação> **by**
<alternativa primária>
<outras alternativas> **else** error

<alternativa primária> ::= <alternativa>

<outras alternativas> ::= <vazia>|<outras alternativas>
**else by** <alternativa>

<alternativa> ::= <lista de comandos>

<teste de aceitação> ::= <expressão lógica>

Exemplo de um bloco de recuperação simples:

```
A: ensure AT
   by        AP: begin
                   ( texto de programa )
               end
   else by   AQ: begin
                   ( texto de programa )
               end
   else error
```

Exemplo de um bloco de recuperação mais complexo:

```
A: ensure AT
   by        AP: begin
                   declare Y;
                 ( texto de programa )
               B: ensure BT
                     by        BP: begin
                                   declare U;
                                     ( texto de programa )
                                   end
                     else by   BQ: begin
                                   declare V;
                                     ( texto de programa )
                                   end
                     else by   BR: begin
                                   declare W;
                                  ( texto de programa )
                                   end
                     else error
                     ( texto de programa )
   else by AQ: begin
                   declare Z;
                 ( texto de programa )
               C: ensure CT
                     by        CP: begin
                                     ( texto de programa )
                                   end
                     else by   CQ: begin
                                     ( texto de programa )
```

```
                                    end
                    else error
            D: ensure DT
                    by          DP: begin
                                    ( texto de programa )
                                    end
                    else error
                end
    else error
```

onde

- A, B e C são rótulos de blocos de recuperação;
- AT, BT, CT e DT são testes de aceitação de seus respectivos blocos de recuperação;
- AP, AQ, BP, BQ, BR, CP, CQ e DP, são rótulos de estrutura de bloco de programa.

Para melhor esclarecimentos, veja RANDELL ( 1975 ), que descreve o esquema mais detalhadamente, além de sugerir uma possível sintaxe para sua descrição formal e mostrar exemplos, acima transcritos. Mais, levanta as preocupações quanto a utilização de tal tipo de estrutura em programação concorrente, diante da possibilidade da ocorrência do efeito dominó, quando da necessidade de se desfazer operações, efetuadas pelo "cache" recursivo do bloco de recuperação, devido a existência de comunicações entre processos. Essas preocupações estão delineadas no sub-capítulo III.5. Quanto aos detalhes do mecanismo de "cache" de recuperação para sua implementação em "software" ( pois a sugestão principal é que seja implementado por "hardware", visando a minimização da sobrecarga imposta ), pode ser visto em HORNING et alii ( 1974 ) que admite que tal sobrecarga imposta é tolerável; ANDERSON et alii ( 1976 ); ANDERSON e LEE ( 1981 ); ( SHRIVASTAVA ( 1978, 1979 ) que estendem a linguagem Pascal concorrente com o conceito de bloco de recuperação; e LEE et alii ( 1980 ) que descreve um, construído para as máquinas da família PDP-11.

## III.4.2 AÇÕES ATÔMICAS

Um dos conceitos sugeridos por vários pesquisadores, preocupados com o problema de confiabilidade em sistemas de computação, se refere em evitar que uma falha se propague de forma descontrolada antes que seja ela detectada. Esse conceito foi denominado por LINDEN ( 1976 ) de "pequenos domínios de proteção", de "isolação de processos" por DENNING ( 1976 ) e "confinamento da falha" por SIEWIOREK ( 1984 ), ( veja item II.2.1 ). Um outro conceito vem a se beneficiar desse, que é "recuperação de erro", pois uma vez detectada a falha e ciente de que esta está sob controle ( está confinada em um domínio bem determinado ), se torna viável na prática, a recuperação do erro que levou a ela. Esse conceito também é citado por DENNING ( 1976 ) e SIEWIOREK ( 1984 ).

Diante disso, várias pesquisas foram desenvolvidas à procura de mecanismos que satisfizessem o objetivo desse conceito, como também o conceito de "recuperação de erro". Tendo, entretanto, na mente, uma preocupação maior, que esses mecanismos não tornassem mais complexos a construção de "software" de grande porte do que é sem eles; isto é, não tornassem mais complexos do que é a programação de um "software" sem a característica de ser tolerante a falhas. Para isso, muitos desse mecanismos são sugeridos que façam parte das características de linguagens de alto nível para programação de sistemas, propondo sintaxes das construções e descrevendo as suas semânticas.

Como mencionado em II.2.2, recorre-se à exploração da noção de ação atômica como um método para implementação dos conceitos acima citados, descritos por LOMET ( 1977 ).

Uma ação atômica é um dispositivo que permite escrever um procedimento garantindo-lhe os mesmos benefícios da atomicidade empregados pelas operações primitivas; isto é, indivisibilidade, não interferência, e seqüenciabilidade estrita. Das três formas sugeridas por LOMET ( 1977 ) para expressar as propriedades importantes de ações atômicas, cita-se a primeira que julgou-se a mais expressiva:

"uma ação é atômica se o processo que a executa não está ciente da existência de outros processos ativos ( não pode detectar mudanças espontâneas de estado ) e nenhum outro processo está ciente das

atividades deste processo ( suas mudanças de estado são escondidas ) durante o tempo que o processo está executando a ação".

A fim de deixar explícito a intenção de que um procedimento é atômico, LOMET ( 1977 ) sugere a seguinte notação:

```
< identificador > : action ( < lista de parâmetros > ) ;
                        < lista de comandos >
                    end ;
```

onde < identificador > e ( < lista de parâmetros > ) são opcionais. Se existir < identificador >, então a ação é do tipo procedimento, cuja semântica é igual ao de um procedimento comum, exceto que ele deve ser executado de forma atômica. Caso não seja colocado um identificador, a ação é uma operação atômica. A lista de comandos pode conter a princípio quaisquer comandos disponíveis na linguagem de programação, inclusive outros procedimentos ações ou operações atômicas.

Assim, ao se desejar definir uma seqüência de operações que devem ser executadas seqüencialmente, de forma indivisível e que não sofrerá interferência de elementos externos a elas, basta confinar tal seqüência pelo par de palavras reservadas "action" e "end" ou pelo par" < identificador > : action ( < lista de parâmetros > )" e "end". Por exemplo:

```
b: action;                    a: action;
   a;                             S;
   .                           end;
   action;
      .
   end;
   .
   a;
end ;
```

Os objetivos de LOMET ( 1977 ) em sugerir esse tipo de notação são, não somente em deixar explícito a intenção desejada de que tal procedimento ou operação sejam executadas de forma atômica, mas principalmente, em deixar a responsabilidade da aquisição e liberação de

recursos computacionais exigidos pela lista de comandos que compõem o procedimento ou a operação ( representados por S e pelos pontos, no exemplo acima ), para o implementador da ação, liberando assim, o programador dessa preocupação.

Para isso, LOMET ( 1977 ) faz duas sugestões. Uma para Sistemas de Multiprogramação em computadores monoprocessadores e uma para Sistemas Multiprocessadores. Para Sistemas de Multiprogramação Monoprocessador, basta que se deixe desabilitado o mecanismo de interrupção, durante a execução de um procedimento ou operação que se quer que seja executado como ação atômica. Assim, a ação retém todo o controle, pegando todos os recursos do sistema durante a sua execução e somente liberando-os ao seu término, quando o mecanismo de interrupção é novamente habilitado. Já em um Sistema de Multiprocessadores, se deseja explorar os recursos eficientemente maximizando a concorrência. Isto requer que somente os recursos, realmente necessários por uma ação atômica durante a sua execução, sejam adquiridos. Os outros processos que desejarem usar esses recursos deverão esperar pela suas liberações. Para isto LOMET ( 1977 ) sugere o seguinte padrão de duas fases: quando um processo quer executar uma ação, ele primeiro adquire todos os recursos compartilhados necessários à execução da ação. O conjunto de recursos é mantido crescente, não liberando nenhum, enquanto existir a necessidade de outros. Esta parte é denominada de fase de crescimento ou fase de aquisição . Uma vez qualquer recurso liberado, nenhum outro poderá ser adquirido e o conjunto de recursos é mantido constantemente decrescente. Esta parte é denominada de fase de retração ou fase de liberação.

Existem várias possibilidades de refinamento dessa estratégia. Uma é permitir que os recursos que somente são examinados não fiquem escondidos dos outros processos. Cabe observar que a estratégia de aquisição e liberação de recursos descrita não se constitui em um algoritmo para gerenciamento de recursos. Um usuário não pode determinar se ele está operando sozinho ou concorrentemente. Não é estabelecido como a contenção é manipulada se a execução deve ser concorrente. Não se descreveu um método para tratar o problema de "deadlock" ( bloqueio perpétuo ). A análise simplesmente descreveu um esquema que um algoritmo de gerência de recursos deve operar.

Note que, dessa forma, a noção de ação atômica provê um meio de se confinar os recursos que são utilizados para a execução de uma

determinada tarefa e, assim, na ocorrência de uma falta ela afetará apenas esses recursos. Mecanismos semelhantes são as regiões críticas e monitores de HANSEN ( 1977, 1978 ) e HOARE ( 1974 ), que LOMET ( 1977 ) tece comentários sobre suas deficiências em comparação com o seu mecanismo.

Naturalmente, é muito interessante, senão indispensável, que erros sejam detectados o mais rápido possível e corrigidos logo em seguida, a fim de se evitar que eles se propaguem, implicando em uma correção maior ou de escapar do controle, não permitindo mais a recuperação do sistema, se tal situação não é desejável ( sistemas tolerantes a falhas ).

Preocupado com o conceito de recuperação de sistema em tempo de execução, LOMET ( 1977 ) sugeriu uma forma de controlar explicitamente a recuperação por meio de um procedimento, denominado de "reset", que é descrito a seguir.

Segundo LOMET ( 1977 ), a recuperação de sistema consiste de duas fases, a saber:

1) rolar o sistema para trás até um estado anterior, admitido válido, desfazendo algum conjunto de ações, que se supõe que inclua aquelas erradas.
2) executar as ações desfeitas na fase anterior que não estavam erradas ( ações conhecidas como tal ).

Para evitar a segunda fase, LOMET ( 1977 ), então, propõe como unidade de recuperação um bloco de recuperação análogo ao proposto por RANDELL ( 1975 ), isto porque o próprio bloco de recuperação é uma ação atômica. Segundo o conceito estabelecido por Randell, é proibido o estabelecimento de comunicação no bloco de recuperação, justamente para isolar o processo que o executa, dos outros e assim restringir a tarefa de desfazer erros apenas daquele que o executa. Desta forma, somente o processo que executa o bloco de recuperação, definido como uma ação atômica, necessita ser restaurado ao estado anterior. A restauração desse processo envolve a restauração, ao estado anterior, somente daquela parte do sistema que é modificada por este processo durante a execução da ação atômica. É desnecessário executar novamente as ações de outros processos, uma vez que elas não foram desfeitas.

Para realizar o "backtracking" ( rolar o sistema para trás ), LOMET ( 1977 ) definiu um mecanismo análogo ao do "cache" recursivo, com as diferenças de que tanto o desejo de desfazer operações, como o

que fazer após isto, são explicitáveis pelo usuário, em seu programa. Esse mecanismo foi denominado de "reset", correspondendo a um procedimento tipo com parâmetros ( passados por valor ), por meio dos quais o usuário pode definir quais ações que ele deseja que sejam executadas após o "backtracking".

Assim, uma unidade de recuperação, sob o conceito do bloco de recuperação, pode ser implementado, por exemplo, como:

```
ensure: action ( accept: boolean function,
                 alternative: array of procedure );
   recover: reset ( j: integer );
      alternative (j);
      if accept then return;
      else
         if j < highbound ( alternative ) then
            recover (j+1);
         else error;
   end recover:
   recover (1);
end ensure;
```

onde "recover" foi definido como um procedimento do tipo "reset" ( local à unidade de recuperação atômica "ensure" ), com o parâmetro especificando qual alternativa será executada após o "backtracking".

No exemplo, o corpo do bloco de recuperação é apenas o comando de chamada de "recover". Assim, uma vez colocado em execução a ação atômica "ensure", a primeira operação a ser executada é o "recover(1). Essa fará o "backtracking" ( no caso, desnecessário, solicitado apenas por questão de algoritmo ) e executará "alternative(1)". Em seguida, será executado o comando "if accept ...", para verificar se os resultados produzidos por "alternative(1)" são aceitáveis. Se forem, então o controle de execução é retornado à ação "ensure", que por sua vez, no caso, encerrará suas atividades. Caso o teste de aceitação falhe, é verificado se existe "alternative(2)" ( "if j < highbound ( alternative ) ..." ). Se existe, então é executado o "recover(2)", que fará o "backtracking", desfazendo as modificações realizadas por "alternative(1)" e executará o "alternative(2)"; e assim por diante ( note que, no exemplo, "recover" é recursivo ). Quando todos os "alternative(i)" falharem, será executado o "else error"; que LOMET

( 1977 ) sugere que seja colocado um outro procedimento do tipo "reset", ou um procedimento do tipo "escape" ( também definido por ele ), para fazer com que o controle volte ao programa que colocou em execução essa ação atômica "ensure" ( naturalmente, com a devida sinalização deste evento ).

### III.4.3 TRATAMENTO DE EXCEÇÕES

As possíveis respostas que podem ser obtidas da execução de um procedimento ( um bloco, um módulo ou uma subrotina ), são classificadas como ( introdução do capítulo 3 de SHRIVASTAVA, 1985 ):

a) esperadas e desejadas,

b) esperadas e não desejadas e

c) não esperadas e não desejadas.

A ocorrência de respostas não esperadas e não desejadas são aquelas provenientes de falhas residuais de projeto e que, portanto, não foram antecipadas, como por exemplo, o colapso do procedimento por ele ter desobedecido suas especificações ( por erro de projeto ou colapso de um componente de "hardware" necessário pelo procedimento ).

Os eventos que fazem com que o procedimento forneça respostas esperadas e não desejadas podem ser, por exemplo: ocorrência de divisão por zero, estouro aritmético, estouro do tempo limite ( "time-out" ), armazenamento ( escrita ) na memória protegida, etc.

De modo geral, ambos esses tipos de eventos indesejáveis, são denominados exceções.

Portanto, quando um procedimento é executado, ou ele termina normalmente ( obtendo-se assim a resposta esperada e desejada ), ou é obtida uma exceção que poderia ser uma resposta esperada e não desejada ou não esperada e não desejada.

Um procedimento em geral poderia conter algoritmos que tratem eventos indesejáveis. Em seu projeto poderia se ter o cuidado de analisar os casos que o impediria de executar os serviços desejados, provendo-o de manipuladores de exceção específicos para tratar tais casos. Para isso, há a necessidade de se detectar a ocorrência de uma exceção. Uma exceção ou é automaticamente levantada ( sinalizada ), geralmente por "hardware", ou é detectada por meio de uma expressão

booleana inserida no procedimento ( ponto de verificação em tempo de execução ).

Se a ocorrência de tais exceções em um procedimento não o afetar no fornecimento de respostas desejadas e esperadas, diz-se que elas foram mascaradas ( tratadas de forma que o procedimento continua a fornecer respostas esperadas e desejadas ). Se entretanto uma exceção não pode ser mascarada pode-se obter como resposta, uma exceção.

Certamente é possível escrever manipuladores de exceção do tipo esperadas mas não desejadas. Mas que estratégia poderia ser utilizada para se tratar eventos não esperados e não desejados?. Uma abordagem sensível é prover um manipulador de exceção "default" ( na falta de ), que desfaça quaisquer efeitos colaterais produzidos pelo procedimento e então, sinalizar tal evento ou executar um procedimento alternativo numa tentativa de mascarar a exceção. A semelhança com o bloco de recuperação não é uma mera coincidência. Realmente é possível definir a semântica de blocos de recuperação em termos de manipuladores de exceção ( MELLIAR-SMITH e RANDELL, 1977 ) - a notação de bloco de recuperação é de fato um meio conveniente de expressar o manipulador de exceção "default". Vê-se a seguir, o porque disso.

Como visto no item anterior, são características do esquema do bloco de recuperação: não fazer nenhuma tentativa para diagnosticar a falha particular que causou o erro; admitir que a extensão dos danos provocados pela falha, fica confinada ao bloco de recuperação; e colocar o estado do sistema como ele estava logo antes da execução do bloco de recuperação e por em execução um de seus blocos alternativos ( isto é feito pelo mecanismo de "cache" de recuperação ). Assim se espera que todas as falhas que possam ocorrer dentro do bloco de recuperação sejam mascaradas. Note, então, que o bloco como um todo pode falhar. Para isso, basta que o erro preceda o bloco de recuperação ou, o seu bloco primário e seus alternativos falhem ou ainda, o próprio teste de aceitação está incorreto. Assim, se o projetista desejar recuperar a falha "on line", ele deverá recorrer a um esquema de detecção de erro mais global e uma forma de recuperação mais drástica ( por exemplo, blocos de recuperação aninhados ).

O tratamento de exceções por meio de blocos de recuperação é então apropriado quando o projetista do sistema não deseja prover um meio individual de tratá-las. Entretanto, para exceções, decorrentes de falhas previstas e cujas conseqüências são conhecidas, é muito mais interessante e prático tratá-las por meio de algoritmos definidos

especialmente para elas ( manipuladores de exceção ). Por exemplo, manipulador para o tratamento da ocorrência de "time-out", manipulador para o tratamento da ocorrência da divisão por zero, manipulador para o tratamento de estouro aritmético, etc. Isto pode muito bem ser constatado e esclarecido nas literaturas sobre linguagens modernas de programação de sistemas ( Ada, CLU, etc ) e sobre "Engenharia de "Software".

MELLIAR-SMITH e RANDELL ( 1977 ), argumentam que, embora os manipuladores de exceções programados tenham seus valores para tratar comportamento indesejável e previsto de componentes de "hardware", usuários, operações de auditoria, etc, são, seguramente, inapropriados para tratar falhas de "software". As falhas de "software" previstas poderiam ser removidas em vez de toleradas. A incorporação desse tipo de manipulador para tratar falhas de "software" aumentaria a sua complexidade, introduzindo assim mais falhas, em vez de ser um meio de competir com as já existentes. Por outro lado, quando usado apropriadamente para falhas antecipadas de outros tipos ( como acima citadas ), eles podem prover um utilíssimo meio de simplificar a estrutura geral do "software" e assim contribuir para a redução de incidências de falhas residuais de projeto.

CRISTIAN ( 1982, 1990 ), investigou alguns conceitos básicos subjacentes às questões de projeto tolerante a falhas e deu uma visão unificada sobre manipulação de exceção programada e manipulador de exceção "default" ( sobre a recuperação aritmética para trás ). Também mostrou que se o intervalo de latência decorrido até a detecção de uma falha for muito grande, o uso de manipuladores de exceção "default" não são convenientes.

### III.4.4 "RETRY BLOCK"

Este esquema, devido a AMMANN e KNIGHT citado em STONE ( 1989 ), propõe uma nova técnica como um meio para a recuperação de erros. O esquema é muito parecido com o do Bloco de Recuperação. A diferença básica está em que, enquanto que aquele utiliza a estratégia da diversidade de projeto, este propõe uma nova, denominada pelos autores de diversidade de dados.

A diversidade de dados é uma estratégia que não altera o algoritmo do sistema, mas sim, os dados sobre os quais o algoritmo

atua. É admitido que existem dados que farão com que o algoritmo falhe e que se eles forem expressos de forma diferente, porém equivalentes ( ou próximos de ), o algoritmo poderá funcionar corretamente.

O "Retry Block" consiste de um algoritmo, um teste de aceitação e um procedimento de re-expressão. A operação desse esquema é a seguinte:

1 - executar o algoritmo;
2 - verificar se os resultados gerados ( saída ) são válidos;
3 - se sim, então continuar com o processamento normal, passando os resultados para o sistema;
4 - se não, verificar se o tempo máximo para execução do algoritmo ( "deadline" ) expirou;
5 - se sim, solicitar uma outra forma de recuperação;
6 - se não, re-expressar os dados e voltar para o passo 1.

Como no esquema do Bloco de Recuperação, existem neste, também, pontos cruciais. Um é o teste de aceitação, cujas dificuldades estão na sua definição correta, para que realmente, ele aceite resultados gerados pelo algoritmo que devem estar de acordo com os esperados pelo sistema ( dificuldades semelhantes ao encontrados no esquema de Bloco de Recuperação ). Um outro ponto, se refere à rotina para re-expressar os dados. O objetivo dessa rotina é tentar converter os dados apresentados em uma forma diferente, que entretanto, preserve a informação neles contidos, na tentativa de que eles ocasionem a geração de resultados pelo mesmo algoritmo, considerados válidos pelo teste de aceitação. Assim, a função da rotina para re-expressar dados será a de mapear o conjunto de dados que geraram resultados inválidos ( conjunto de dados inválidos ), para um conjunto de dados válidos; que, naturalmente, inclui o conjunto de dados idênticos, pois pode ser que a geração de resultados inválidos tenha sido provocada por falta transiente do "hardware".

### III.4.5. PROGRAMAÇÃO N-VERSÕES

A pesquisa do uso do projeto de diversidade para se obter tolerância a faltas neste esquema de programação N-Versões, foi primeiro descrito por CHEN e AVIZIENIST ( citado em STONE, 1989 ).

Este esquema não é uma técnica de recuperação de erro para trás ou para a frente ( tratamento de exceções ). A abordagem é

análoga ao esquema de redundância de "hardware", discutida em vários artigos que propõem formas de se impor confiabilidade em "hardware" por redundância ( veja STONE, 1989 ). A técnica é tentar mascarar a ocorrência de falhas em um módulo que se deseja que seja tolerante a falhas, por meio de sua redundância. Dessa forma, o esquema requer o seguinte: um conjunto de implementações diferentes de uma mesma especificação de um módulo ( por exemplo, procedimento ou programa que se deseja que seja tolerante a falhas ) e um procedimento de votação. Cada implementação de módulo deverá ser projetada sem fazer referência aos outros. Não existe requisitos para que os módulos sejam programados em uma mesma linguagem, como também, que sejam executados em uma mesma máquina, desde que seus resultados sejam colocados à disposição do mecanismo de votação. A figura abaixo ilustra o esquema.

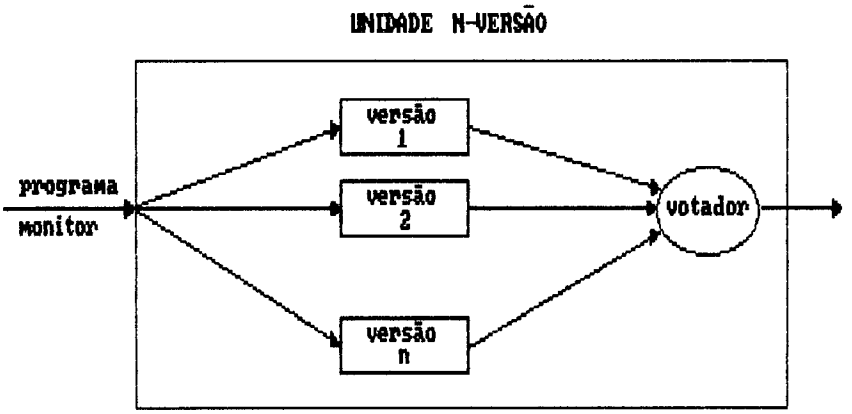



Figura III.4 Esquema de Programação N-Versão

O funcionamento dessa forma de tolerância a faltas é muito simples ( uma de suas vantagens ). O programa principal que está em execução, conhecido como o monitor, coloca em execução todas as versões da implementação da unidade N-Versões em paralelo e apresenta os resultados de cada versão ao votador para serem comparadas. Desse modo, as tarefas do monitor devem ser as seguintes:

a) colocar em execução cada versão, componente da unidade N-Versões;
b) esperar o término da execução de todas as versões;
c) executar o mecanismo de votação.

Não há a necessidade de que todas as versões sejam executadas em paralelo, elas podem ser executadas seqüencialmente. Entretanto, cada versão deve ser executada de forma independente das outras, isto

é, atomicamente. Não pode existir interações entre elas e devem ser impedidos de mascarar mudanças do estado global e das saídas. O conjunto de entradas ( os dados e o estado corrente ) para cada versão devem ser idênticas. Para a coordenação da execução das versões podem ser usadas primitivas simples de sincronização, por exemplo as operações sobre semáforo - "wait" e "signal", como segue. Cada versão espera por um sinal do monitor antes de entrar em execução. O monitor então espera o término de cada versão. Quando uma versão termina, ele sinaliza o monitor usando a primitiva "signal". O problema dos tempos diferentes de execução é mais sério se é considerada a possibilidade da existência de processos desertores. Uma forma para detectar a ocorrência desse fato, é por meio da utilização de um temporizador cão de guarda que define um tempo limite ( "time-out" ) para quaisquer versões incompletas.

O mecanismo de votação é controlado pelo monitor e sua função é comparar as saídas de cada versão e decidir se homologa ou não os resultados da unidade N-Versões. Tal mecanismo é fornecido pelo programador, de maneira semelhante ao teste de aceitação do Bloco de Recuperação, e pode levar em consideração discrepâncias permissíveis das saídas. Um exemplo de discrepância está na representação de números em ponto flutuante. Dois sistemas podem produzir valores muitos próximos limitados somente pelo "hardware" que os produziram. Para permitir que um consenso seja encontrado, a despeito dessas diferenças, requer-se uma votação inexata. Uma verificação de campo é proposto como em exemplo simples de mecanismo de votação inexata.

Maiores detalhes e variantes, podem ser encontrados, por exemplo, em ANDERSON e LEE ( 1981 ), MANCINI e KOUTNY ( 1986 ) e MANCINI e PAPPALARDO ( 1987, 1988b ).

### III.4.6 BLOCO DE RECUPERAÇÃO DE CONSENSO

O bloco de Recuperação de Consenso, seguindo STONE ( 1989 ), é um esquema que combina as técnicas utilizadas na Programação N-Versões e no Bloco de Recuperação. Como se delineia a seguir, percebe-se que o esquema procura solucionar ou pelo menos simplificar os problemas encontrados nas duas técnicas referidas, que entretanto, apresenta outros.

O esquema requer o projeto e a implementação de N versões independentes do algoritmo que são ordenados ( como no Bloco de Recuperação ) para, tanto executar o serviço, como para atender o requisito de confiabilidade desse serviço. De forma análoga ao esquema de programação N-Versões, todas as versões do algoritmo são executadas ( preferencialmente em paralelo ) e seus resultados submetidos ao votador, que os homologa como saída se dois resultados forem iguais. Caso não encontre um par de resultados, os da melhor versão são submetidos a um teste de aceitação. Se esse falhar, os resultados da próxima versão são então submetidos. Esse processo continua até que os resultados de uma versão sejam aceitos ou o bloco como um todo falha.

Observa-se que esse esquema se protege contra processos desertores, pois basta que os resultados de duas versões se casem. Entretanto, se nenhum par se casar o votador pode esperar por desertores, não garantindo, dessa forma, uma proteção total. Além disso, mesmo que seja encontrado um par, como não existe nenhuma indicação que os resultados sejam submetidos ao teste de aceitação, pode ser que falhas correlatas nas duas versões gerem resultados iguais e, assim, usar como saída do bloco, resultados errados. Cabe observar ainda que, a hipótese da não existência de faltas comuns é muito forte. Entretanto, segundo Stone, foram desenvolvidos modelos de confiabilidade por Scott, Gault e McAllister, para três esquemas, o Bloco de Recuperação, a Programação N-Versões e o Bloco de Recuperação de Consenso. Comparando-os, o último mostrou-se ser o mais confiável.

Nota-se nesse esquema as seguintes vantagens em relação aos dois anteriores. A saber:

a) o teste de aceitação é processado caso não se encontre nenhum par de versões que gerem resultados iguais; naturalmente, a palavra iguais deve ser interpretada como pelo votador do esquema de Programação N-Versões. Dessa forma, o teste de aceitação fica colocado em segundo plano;

b) o problema que processos desertores geram, fica amenizado pelo mesmo motivo anterior; é apenas dependente da implementação do esquema, se procura ou não se proteger completamente, com citado anteriormente.

c) nesse esquema procura-se ordenar as versões do algoritmo segundo o esquema do Bloco de Recuperação. Assim, as primeiras versões, provavelmente, sempre fornecerão resultados condizentes,

garantindo que muito raramente o teste de aceitação será executado, como também, o votador não precisa esperar pelos resultados de todas as versões, para homologar a saída do bloco. Essas vantagens, já estão incluídas nas duas primeiras, porém, explicitou-as dessa forma para tornar mais claro que elas, talvez, mostrem que o esquema não garante uma alta confiabilidade. Entretanto, dá para perceber que ele deve exigir, na média, muito menos tempo de processamento para validar resultados desejados, do que os dos esquemas de Programação N-Versões e Bloco de Recuperação.

## III.5 ESTRUTURAÇÃO DE SISTEMAS DISTRIBUÍDOS TOLERANTES A FALHAS

Até agora foram discutidas as técnicas para estruturação de Sistemas Distribuídos Tolerantes a Falhas existentes na literatura que, entretanto, se limitaram a especificidades dos conceitos e técnicas para tolerar falhas, como por exemplo, o bloco de recuperação, ações atômicas, etc. Todas elas, praticamente, utilizando como base as clássicas estruturação de "software" em camadas e a programação estruturada.

Discute-se e descreve-se a seguir, a proposta de RANDELL ( 1984 ) - "Fault Tolerance and System Structuring" -, cujo objetivo principal foi definir uma metodologia geral para se projetar sistemas de computação tolerante a falhas, concentrando-se sobre as questões de estruturação em vez dos projetos particulares de algoritmos. Isso porque, com os sistemas de computação que têm que satisfazer especificações complexas de demanda, os níveis de confiabilidade que podem ser impostos dependerá crucialmente do quanto o projeto do sistema pode ser mantido simples. Assim, no mínimo, a estruturação cuidadosa é tão importante quanto são os algoritmos engenhosos para realizar com sucesso o projeto de sistema tolerante a falhas ( veja também RANDELL, 1987 ).

Basicamente, RANDELL ( 1984 ), propõe três formas de estruturação de sistemas. A primeira consiste em estruturar o sistema a partir de componentes denominados de "componentes idealizados tolerantes a falhas". A segunda se baseia na utilização de tais componentes, quando esses são computadores completos ( "hardware" mais o "software" ), para a construção de sistemas

multiprocessadores; tal esquema foi denominado de "estruturação recursiva" de sistema. A terceira, discute uma generalização do uso do conceito de ações atômicas para a estruturação de recuperação de erros de forma antecipada e para trás em Sistemas Distribuídos.

A seguir resume-se essas três formas de estruturação de sistemas.

## III.5.1 COMPONENTES IDEALIZADOS TOLERANTE A FALHAS

Seguindo RANDELL ( 1984 ), tanto os sistemas, como os seus componentes, podem ser considerados como realizadores de operações a fim de fornecer respostas aos pedidos feitos. Do ponto de vista de um componente, as falhas podem ser agrupadas em três categorias, a saber:

(a) falhas dentro do próprio componente;

(b) falhas nos sub-componentes ou componentes co-existentes que um componente pode utilizar; e

(c) pedidos faltosos de serviço a componente, feitos pelo seu ambiente, isto é, do componente que o encapsula ou de componentes co-existentes com quem ele interage.

Um componente idealizado tolerante a falhas, deve ser esquematizado de modo a lidar com esses tipos de falhas, de forma a ( veja figura III.5 ):

- reportar ao seu ambiente os pedidos faltosos ( falhas do tipo (c) ), emitindo sinais de exceção de interface;
- procurar tratar todas as falhas internas, por meio de manipuladores de exceções ( falhas do tipo (a) ), levantando exceções locais;
- procurar tratar as falhas de seus sub-componentes ou co-existentes com quem ele interage ( falhas do tipo (b) );
- reportar ao seu ambiente as falhas que ele não consegue tratar, emitindo sinais de exceção de falha;
- emitir pedidos de serviços aos componentes co-existentes com que ele deseja interagir; e
- receber respostas normais ou anormais a esses pedidos, sendo que as anormais, através de sinais de exceção de interface ou de falha.

Dessa maneira, o esquema implica na minimização das hipóteses sobre os tipos de falhas que não podem ocorrer e que tipos de mascaramento de falhas devem ser construídos, tomando-se o cuidado

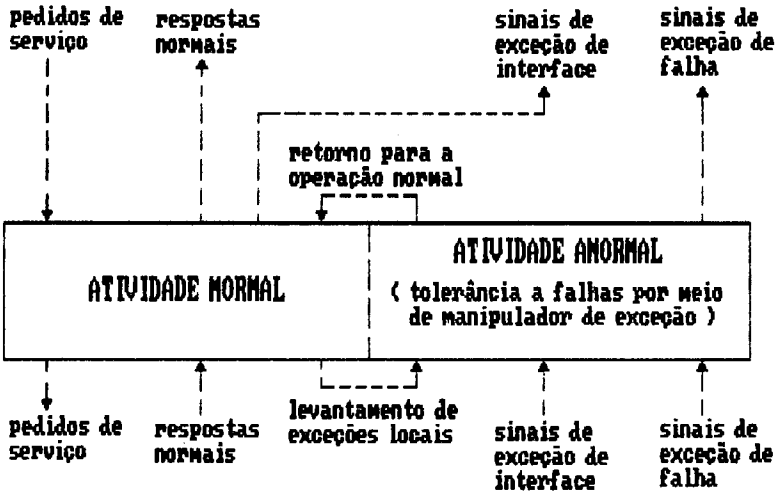



Figura III.5: Um componente idealizado tolerante a falhas

de que alguma falha não invalide a estrutura planejada dos componentes e de suas intercalações.

Com tal esquema de estruturação é possível, e realmente desejável, especificar completamente a interface entre cada componente e seu ambiente. Isso permite que o projeto de um componente seja baseado apenas na especificação da interface e ser compreendido independentemente de seu ambiente, mesmo com respeito a questões de tolerância a falhas.

Assim, a tarefa mais importante de um projetista de sistema para utilizar esse esquema de estruturação, será a de decidir que tipos de falhas podem ocorrer e quais não podem, e quais são as formas de mascaramento de falhas que realmente serão eficientes. Isto é, decidir que exceções devem ser definidas e que tratadores de exceções devem ser providos. Por exemplo, se um componente usa a técnica da redundância modular tripla para mascarar faltas de um sub-componente, poderá, em princípio, ter um meio de reportar uma falha de exceção ao seu ambiente, se não existir concordância de resultados, pelo menos por dois módulos.

## III.5.2 ESTRUTURAÇÃO RECURSIVA DE SISTEMAS

A generalidade da noção de componente idealizado tolerante a falhas, permite que ela seja aplicada a muitos diferentes aspectos e níveis de projeto de sistemas de computação. Assim, a idéia básica do esquema de estruturação recursiva de sistemas, é a de utilizar

computadores completos como tais componentes para a construção de sistemas de maior porte, por exemplo, um Sistema Distribuído. Essa técnica é expressa pela seguinte regra:

*"Um Sistema Distribuído deverá ser funcionalmente equivalente a sistemas individuais dos quais ele é composto".*

Em outras palavras, se quer fazer com que a interface externa de cada computador componente se case exatamente com as especificações do sistema como um todo, com a preocupação de que a facilidade de nomeação dos recursos computacionais de cada um, deve independer de seu contexto arquitetural; isto é, se funcionará isoladamente ou como componente de um sistema maior.

### III.5.3 AÇÕES ATÔMICAS

As duas técnicas para estruturação de sistemas tolerantes a falhas anteriores, impõem uma estruturação estática dos componentes do sistema. Isso se deve à utilização, na forma tradicionalmente usada, de manipuladores de exceção como tratadores de falhas que implica ou na continuação normal do fluxo de processamento, ou na execução da exceção levantada que, por sua vez, posteriormente dá continuidade normal de processamento, ao executar um algoritmo alternativo, ou apenas transfere a forma de tratamento para um componente do sistema, mais adequado para tal. De qualquer forma, nota-se, assim, que ou o serviço pedido foi plenamente executado ou foi levantada uma exceção e, preferencialmente, nada foi executado. Daí, a importância salientada de que os projetistas devem definir muito bem os tipos de falhas que podem e quais não podem ocorrer. Entretanto, diante da grande gama de falhas que podem surgir em Sistemas Distribuídos e diante do desejo de que haja o maior paralelismo possível na execução das tarefas do sistema, muitas falhas, mesmo que previsíveis, podem ser detectadas apenas após a execução da tarefa, o que exigirá que os efeitos residuais deixados pela tarefa devam ser desfeitos, ao se desejar continuar com o processamento normal do sistema. Porém, desfazer efeitos de operações executadas pode implicar em desfazer-se de outras operações definidas em outros componentes. Assim, um componente de sistema desse porte pode acabar por se tornar muito grande e, pior,

tendo que ter um comportamento síncrono - daí a imposição de estruturação estática mencionada.

Diante disso, RANDELL ( 1984 ), propõe o uso do conceito de ações atômicas para estruturar tanto recuperação de erros antecipáveis, quanto as não antecipáveis ( recuperação para trás, desfazendo efeitos de operações realizadas ) em sistemas assíncronos, para permitir que os seus componentes sejam mantidos de forma mais simples possível. Para tanto, é necessário uma especificação precisa dos estados inicial e final de cada ação atômica que seja independente de quaisquer atividades assíncronas, dentro ou fora dela, para que se possa projetar manipuladores de exceções gerais.

A tentativa da proposição de tal esquematização para a estruturação de sistemas, teve como base a seguinte: "o levantamento de uma exceção dentro de uma ação atômica tolerante a falhas, requer a aplicação de computação anormal e mecanismos para implementar as medidas de tolerância a falhas. Se as medidas de recuperação se sucedem, a ação atômica deverá produzir os resultados normalmente esperados. As ações atômicas que explicitamente retornarem um resultado anormal, as fazem somente com a concordância de todos os seus componentes. Assim, associou-se o contexto de manipuladores de exceção com ações atômicas".

O esquema, resumidamente, é o seguinte ( veja figuras III.6 e III.7 ):

- associação do contexto do manipulador de exceção com ações atômicas;
- ações atômicas podem ser conter outras ações atômicas;
- se é levantada uma exceção dentro de uma ação atômica interna, então medidas para tolerar a falha devem ser aplicadas ( atividade anormal );
- caso a exceção não possa ser tratada internamente, a ação atômica deve sinalizar tal exceção para a ação atômica que a contém;
- se um ou mais componentes de uma ação atômica levantam uma exceção, essa deve ser única para a ação que a contém;

Observa-se que a construção desses tipos de ações atômicas é ainda especulativa, pois ainda devem ser inventadas muitas regras para os manipuladores de exceção para que elas não invalidem os conceitos básicos de ações atômicas.

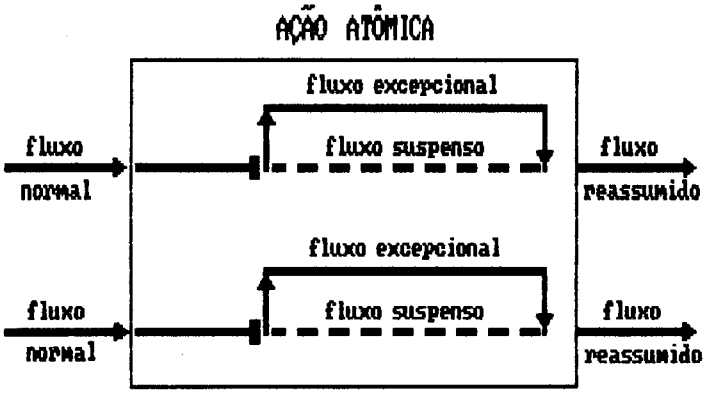



Figura III.6: Recuperação de erro com sucesso em uma ação atômica

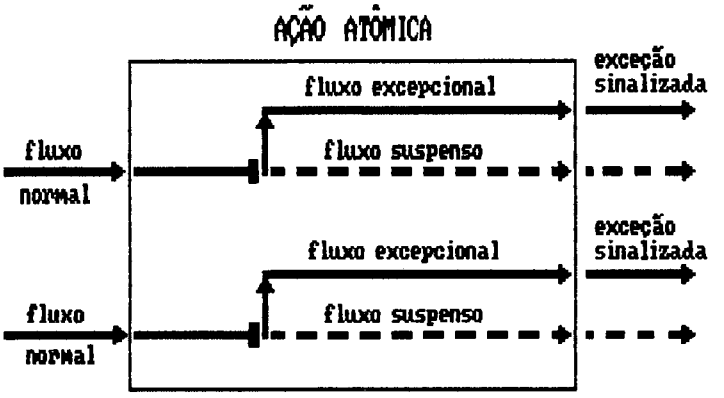



Figura III.7: Sinalização de uma exceção de uma ação atômica

# CAPÍTULO IV

## RECUPERAÇÃO DE SISTEMAS DISTRIBUíDOS

Recuperar um sistema, como já descrito, significa, rolar o sistema para trás até um estado anterior, admitido válido, desfazendo-se de algumas operações que se admitem que incluam aquelas erradas.

Para tanto, em termos gerais, existem duas abordagens. Uma denominada abordagem planejada e outra denominada abordagem não planejada.

Na abordagem planejada, os pontos de recuperação e seus descartes são coordenados pelos processos envolvidos em uma dada ação atômica. Na outra, os processos interativos não se preocupam com essa coordenação. Quando é detectado um erro, é necessário construir dinamicamente um estado consistente de pontos de recuperação, denominado de "linha de recuperação". Tal construção, corresponde à procura por uma ação atômica ainda incompleta.

Diante da complexidade das soluções propostas para a determinação de linhas de recuperação, é descrito um mecanismo de recuperação de processos que evita tal necessidade. Para tanto, ele impõe a restrição de que os processos devem ser determinísticos; isto

é, um processo recuperado, terá o mesmo comportamento que teve até o instante da ocorrência de sua falha. Note, assim, que como o processo diretamente afetado pela ocorrência de alguma falha, cujo tratamento exigiu a sua recuperação para um estado anterior a esse evento, executará exatamente a mesma seqüência de operações, esse procedimento evita que os processos com quem ele interagiu nesse trecho, tenham que ser também recuperados. Dessa forma, tal mecanismo pressupõe que os processos estão corretos e se utiliza de um sistema de troca de mensagens que execute, além de suas tarefas normais, algumas especificamente para esses propósitos.

Além disso, independentemente da necessidade de se rolar o sistema para trás a fim de recuperá-lo, pode ser preciso que alguns processos migrem de um nodo operador para um outro, objetivando encontrar os requisitos de confiabilidade exigidos ( ou impostos ) pelo sistema.

Uma questão importante sobre tolerância a falhas em Sistemas Distribuídos, se refere aos mecanismos de troca de mensagens entre processos remotos que, para competir com as falhas do meio de comunicação, utilizam a retransmissão de mensagens e/ou desistem, levantando tal exceção, para que o processo fique livre para continuar com o seu processamento. Esses esquemas de tolerância a falhas no sistema de troca de mensagens pode criar computações órfãs, que se não forem devidamente expurgadas, podem gerar eventos indesejáveis como a reexecução de operações não idempotentes, por exemplo.

A seguir descreve-se com maiores detalhes essas abordagens, suas aplicações, os cuidados que se deve tomar para se migrar um processo e o problema dos órfãos.

## IV.1 ABORDAGEM PLANEJADA DE RECUPERAÇÃO DE SISTEMAS DISTRIBUÍDOS

### IV.1.1 ABORDAGEM PLANEJADA DE RECUPERAÇÃO DE SISTEMAS CENTRALIZADOS

Um método para recuperação de sistemas, quando da detecção de um erro, é dita planejada, quando o estabelecimento dos pontos de recuperação e seus descartes são coordenados pelos próprios processos que estão envolvidos em uma dada ação atômica. Por exemplo, quando a

recuperação é baseada nos conceitos de bloco de recuperação ( HORNING et alii, 1974; RANDELL, 1975 e ANDERSON e LEE, 1981 ) ou das ações atômicas como descritas por LOMET ( 1977 ).

Para se desfazer de operações, HORNING et alii ( 1974 ) descreveu um mecanismo que o denominou de "cache" de recuperação. Em seu conceito de bloco de recuperação, tal mecanismo é implicitamente acionado, quando o teste de aceitação do bloco fracassa. Já no conceito de ações atômicas ( LOMET, 1977 ), tal mecanismo é explicitamente solicitado; que para isso, foi sugerido que ele se constituísse em um procedimento.

Tanto o conceito de bloco de recuperação, como as ações atômicas daquelas formas descritas, não se preocupam em diagnosticar as causas das falhas. Procura por meio de procedimentos alternativos mascará-las. Essa forma de mascaramento, é adequada quando o tipo das falhas não são antecipáveis, como, aliás, principalmente, o conceito de bloco de recuperação, foi proposto. A detecção da ocorrência de uma falha é feita por meio da verificação de uma asserção, que resultará verdadeira se tudo correu como planejado, caso contrário, resultará falsa, indicando a ocorrência de uma anormalidade qualquer com o procedimento executado. Já quando as falhas são antecipáveis, é melhor construir procedimentos específicos para os seus tratamentos ( MELLIAR-SMITH e RANDELL, 1977 ).

Naturalmente, é inviável construir um sistema, do tipo Sistema Operacional, constituído de um único bloco de recuperação ou ação atômica. Pois, isso implicaria em se implementar vários sistemas alternativos, cada um tentando mascarar uma falha. Isso ocorreria se fosse considerada a possibilidade de se poder construir um sistema, cujo processamento fosse puramente seqüencial. Algo parecido com isso é proposto por RANDELL ( 1984 ), visto no item III.5.

Ignorando, no momento, a proposta de RANDELL ( 1984 ), já é bem conhecida a técnica de se construir um Sistema Operacional por meio de processos, que podem ser executados em concorrência, na tentativa de se otimizar a utilização dos recursos de "hardware". Assim, normalmente, um Sistema é composto de vários processos que interagem ( sincronizam e trocam informações ), para atingir os objetivos definidos para ele.

Assim, pode parecer que se construir cada processo por meio de um bloco de recuperação ou ação atômica, resolveria o problema de recuperação de sistema em caso de ocorrência de alguma falha

qualquer. Entretanto, diante da existência da interação entre eles, existe a possibilidade de que, ao se desfazer as operações realizadas por um processo ( por motivo da detecção de uma falha ), seja necessário desfazer as operações de um processo com quem ele interage, que por sua vez exige o mesmo para um outro e assim por diante, podendo até acontecer de acabar desfazendo todas as operações feitas até então, num verdadeiro efeito dominó ( RANDELL, 1975 ). Para se evitar esse problema, RANDELL ( 1975), afirma que basta remover uma das seguintes circunstâncias, que se existirem em combinação se está sujeito ao efeito dominó:

(1) As estruturas de bloco de recuperação dos vários processos estão descoordenados, e não levam em consideração as interdependências causadas por suas interações.

(2) Os processos são simétricos com respeito à propagação das falhas - qualquer processo de um par de processos que interagem, pode implicar no recuo do outro.

Para tal, RANDELL ( 1975 ) propõe uma técnica para estruturação de processos que interagem entre si, a qual ele denominou de "conversação", para tratar a questão (1). Quanto à questão (2), ele propõe a construção de sistemas em camadas interpretativas ( cada camada é uma máquina virtual ), com cada uma oferecendo operações confiáveis ( tolerantes a falhas ), à camada logo acima dela. Assim, cada máquina virtual oferece uma interface tolerante a falhas à máquina de nível superior, que também deverá ser tolerante a falhas, e assim por diante, até se atingir a máquina virtual da qual os usuários se servirão, para execução de suas tarefas ( veja item III.2 ).

A técnica da conversação, basicamente, se resume em encapsular dois ou mais processos que interagem entre si de forma a se tornarem mutuamente dependentes em seus progressos, em um único bloco de recuperação, sob as seguintes restrições, para realmente evitar o efeito dominó ( veja figura IV.1 ):

a) qualquer um dos processos encapsulados não pode comunicar-se com um processo não pertencente a essa conversação;

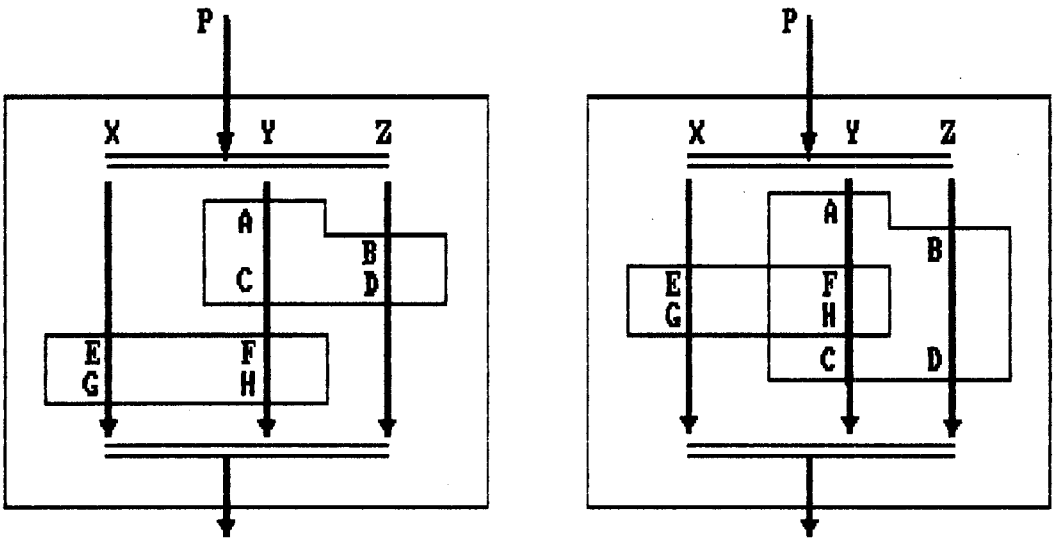
b) no final da conversação, todos os processos devem satisfazer seus respectivos testes de aceitação e nenhum pode prosseguir antes que todos tenham sido validados;

c) as conversações podem ser aninhadas, desde que estritamente aninhadas.

Com isso, essa técnica, provê uma estrutura de recuperação que é comum ao conjunto dos processos.

A outra proposta de RANDELL ( 1975 ), para se evitar o efeito dominó, é estruturar um sistema em camadas, cada uma se constituindo em uma máquina virtual tolerante a falhas. Com isso, impõe-se a assimetria da propagação de uma falha, pois, cada máquina virtual utiliza operações tolerantes a falhas oferecidas por uma máquina virtual logo subjacente, para implementar operações, também, tolerantes a falhas para a construção de uma máquina de nível mais alto.

Essa técnica, além de evitar o efeito dominó, é a mais conhecida para a estruturação de Sistemas Operacionais, para competir com




(a) (b)

- As figuras geométricas fechadas ( "retângulos" ), representam blocos de recuperação que mantêm conversações;
- X, Y, Z são processos paralelos, criados dentro do bloco de recuperação de P;
- A, B, E e F, são pontos de recuperação de blocos de recuperação;
- C, D, G e H, são as saídas de blocos de recuperação.

a) Processos paralelos com conversações que provêem blocos de recuperação para comunicação local.

b) Exemplo de conversações inválidas, por não estarem estritamente aninhadas.

Figura IV.1: Conversação - uma técnica para evitar o efeito dominó

a complexidade inerente ao "software" desse porte, com ou sem aspectos de tolerância a falhas. Obviamente, um sistema com características de tolerância falhas, é muito mais complexo. Interessados na construção de Sistemas Operacionais tolerantes a falhas, várias propostas foram colocadas. Por exemplo, a de ANDERSON e SHRIVASTAVA ( 1978 ) e ANDERSON e LEE ( 1979, 1981 ), que será a seguir delineado.

Anderson e SHRIVASTAVA ( 1978 ) e ANDERSON e LEE ( 1979, 1981 ), propõem um modelo de recuperabilidade em Sistemas Multiníveis. Nesse, ele considera os mecanismos de recuperação para tornar as seguintes características disponíveis, para os programas que serão executados sobre uma interface:

i ) A interface suporta tanto objetos recuperáveis como irrecuperáveis;

ii ) Pontos de recuperação podem ser estabelecidos que garantam que o estado corrente dos objetos recuperáveis da interface ( no mínimo conceitualmente ) sejam registrados de modo que eles possam ser restaurados se necessário;

iii) Pontos de recuperação podem ser descartados com o efeito de que a informação mantida para a recuperação para aqueles pontos é descartada.

Naturalmente, em uma interface, objetos recuperáveis são aqueles para os quais são providos suas recuperações e, objetos irrecuperáveis são aqueles para os quais o estado de restauração não está disponível ou não é apropriada para tal. Por exemplo, objetos usados para modelar os efeitos do ambiente externo, como um relógio externo.

Em seu modelo, Anderson, introduz terminologias que serão utilizadas daqui por diante, que são:

- Ponto de recuperação ativo: desde quando ele é estabelecido até o seu descarte;
- Região de recuperação: esse termo é usado para se referir ao período para o qual um ponto de recuperação está ativo;
- Ambiente de recuperação: é aquele conjunto de objetos recuperáveis que estão disponíveis em uma interface no instante em que é estabelecido um ponto de recuperação. Por exemplo, em um programa, o ambiente de recuperação consistiria do conjunto das variáveis recuperáveis que existiam no instante do estabelecimento de um ponto de recuperação, porém excluiria aquelas criadas subseqüentemente.

Anderson procura em seu modelo, não impor restrições que acabem por particularizar a sua aplicação. O modelo permite que um programa contenha em um instante vários pontos de recuperação ativos e permite que regiões de recuperação se sobreponham, totalmente ou parcialmente, como ilustrados pelas figuras IV.2.

Diante da existência de Sistemas Multiníveis construídos por meio de camadas interpretadoras ou camadas como extensões do interpretador básico, aqueles que tratam dos detalhes do "hardware" ( veja item III.2 ), são discutidos os problemas de se prover recuperabilidade em tais tipos de interfaces.

Em Sistemas Multiníveis compostos de somente interpretadores, em cada interpretador ( cada camada ) deve-se incluir programas e suas estruturas de dados, de modo que toda a informação necessária para a recuperabilidade daquela interface seja mantida. Tais programas são denominados de programas de recuperação que, de modo geral, se preocupam com as seguintes ações: registrar os dados de recuperação; executar a recuperação e realizar o encerramento da recuperação.

Dessa forma, qualquer ambiente de recuperação em um nível do sistema, é disjunto daquele que o suporta. Um ponto de recuperação ativo em um nível é completamente independente daqueles de outros níveis.

Nesse esquema, os objetos irrecuperáveis, providos por uma interface, devem ser tratados pelos programas que os utilizam, que realmente sabem como registrar os dados para suas recuperações, na detecção da ocorrência de alguma falha, quando de suas utilizações.

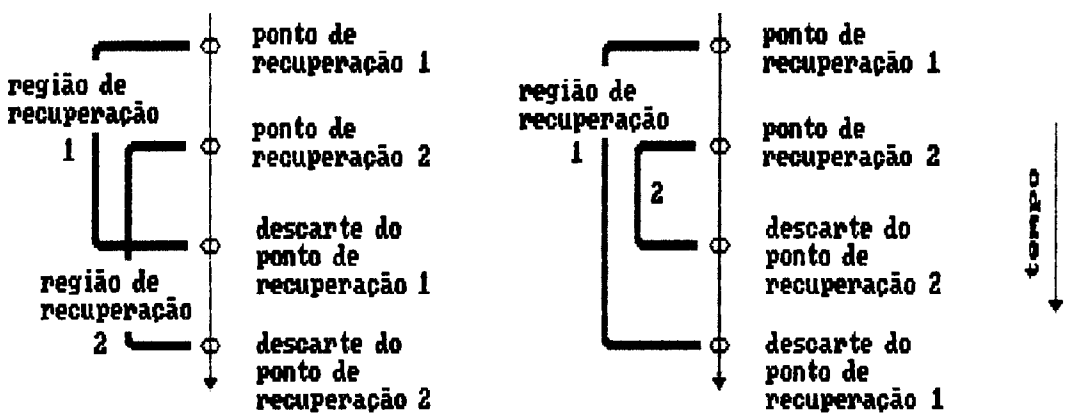



Figura IV.2: Múltiplos pontos de recuperação

O modelo de recuperabilidade para a outra forma de construção de Sistema Multinível, de modo geral, se resume no seguinte: o interpretador básico é implementado como no esquema anterior, dando suporte tanto a objetos recuperáveis como os irrecuperáveis. Cada extensão, também, é provido com características de recuperação para os seus objetos abstratos, a fim de inclui-los à interface existente.

Para se estender tais novos objetos abstratos a um interpretador existente, existem duas formas, a saber: estender de modo que eles sejam considerados como partes do interpretador subjacente ( disjuntos de qualquer programa que os utiliza ), ou para serem considerados como componentes inclusive dos programas que os utilizam. Como conseqüência, existe a distinção de tratamento dos objetos recuperáveis por uma extensão; especificamente, se eles são ou não considerados como estando dentro do ambiente de recuperação de um programa que os utiliza.

Quando a extensão é para ser considerada como parte do interpretador subjacente, o esquema para o tratamento dos objetos recuperáveis foi denominado de "esquema disjunto de recuperação". Para o outro tipo de extensão, o esquema recebeu a denominação de "esquema inclusive de recuperação".

## ESQUEMA DISJUNTO DE RECUPERAÇÃO

Nesse esquema, o ambiente de recuperação de um programa é disjunto daqueles de qualquer extensão que o suporta. Isto é, cada extensão é responsável pela recuperação de todos os objetos por ela mantidos. Assim, quando um programa estabelece um ponto de recuperação, o esquema deve garantir que o ambiente de recuperação daquele ponto somente inclua os objetos abstratos disponíveis sobre a interface subjacente ( o interpretador básico ), e não aqueles das extensões.

O esquema não proíbe que uma extensão estabeleça seus próprios pontos de recuperação. Nessa situação, qualquer objeto recuperável usado pela extensão dentro de uma região de recuperação se comportaria normalmente. Quando um ponto de recuperação local for descartado, todos os dados de recuperação mantidos para aquele ponto deverão ser descartados.

De um modo geral, a recuperação em um Sistema Multinível implementado por extensões do interpretador com o esquema disjunto de

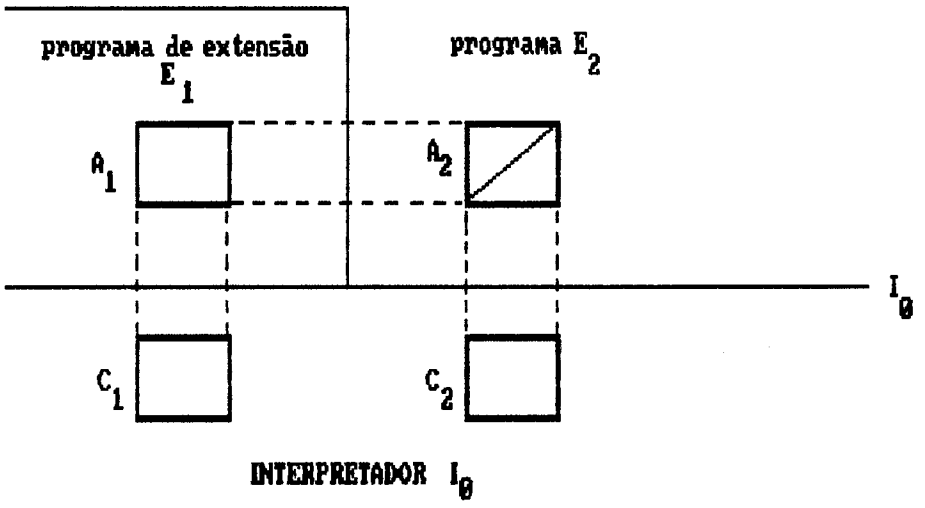
recuperação, é como segue. "Quando da detecção de um erro em um programa $E_i$, o interpretador subjacente restaura todos os objetos que ele mantém para $E_i$. O interpretador, então, sinaliza todas as extensões que poderia ser diretamente chamado por $E_i$ ( do conjunto $E_1$, $E_2$,..., $E_{i-1}$ ), de modo que elas realizem a recuperação dos objetos que mantêm diretamente para $E_i$. Seguindo o término dessas ações, o programa $E_i$ estará recuperado e pode reiniciar quando necessário. Conceitualmente, o interpretador tem que, também, sinalizar todas as extensões diretamente acessíveis para $E_1$, $E_2$,...,$E_{i-1}$, sempre que $E_i$ cria ou descarta um ponto de recuperação, de modo que as extensões possam, de maneira similar ao interpretador subjacente, registrar ou encerrar os dados de recuperação necessários para o programa $E_i$".

Uma característica significativa desse esquema é que o comportamento de uma extensão em relação a objetos recuperáveis e irrecuperáveis, é uniforme. Todavia, o esquema pode incorrer em uma deficiência se for necessário reconsiderar objetos recuperáveis pelas extensões ( o que ocorre em um sistema puramente interpretativo ).

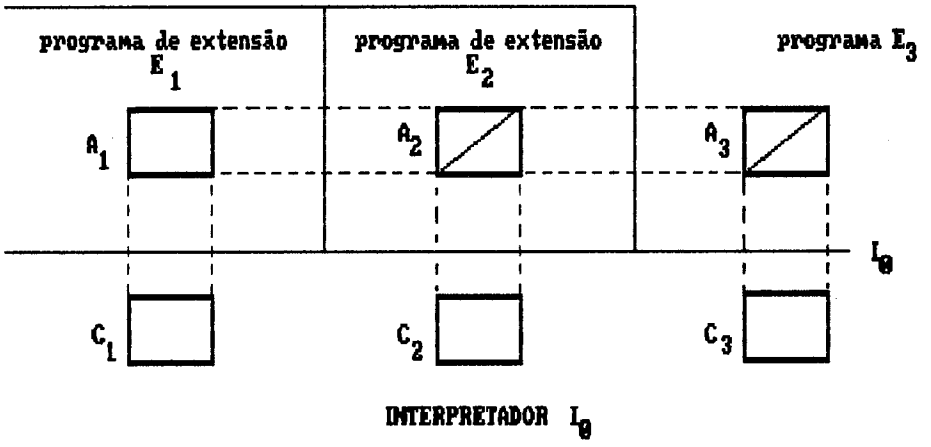
## ESQUEMA INCLUSIVE DE RECUPERAÇÃO

Esse esquema decorreu da procura por uma alternativa para se evitar a deficiência do esquema anterior. Os objetos recuperáveis usados em uma extensão são considerados como pertinentes ao programa que os utiliza, e são automaticamente recuperados quando ele é restaurado ao seu ponto de restauração. Esse esquema foi denominado de esquema de recuperação inclusive. O esquema é o seguinte, ilustrado pela figura IV.3.

Pela figura IV.3 ( a ), os objetos recuperáveis de $A_1$, são considerados como pertinentes ao ambiente de recuperação do programa $E_2$. Conseqüentemente, quando $E_2$ estabelece um ponto de recuperação, quaisquer dados de recuperação gerados para os objetos $A_1$ recuperáveis, serão mantidos com os dados para recuperação associados com $E_2$. Dessa forma, quando o programa $E_2$ é recuperado, a extensão $E_1$ pode admitir, quando ativado por $I_0$, que todos os objetos recuperáveis utilizados por ela, serão automaticamente restaurados; $E_1$, portanto, precisa somente alterar, quando necessário, os objetos irrecuperáveis usados no estado concreto. Como no caso disjunto, o interpretador subjacente $I_0$, terá que sinalizar a extensão, para obter essa recuperação.




( a ) mapeamento de espaço de dados abstratos




( b ) mapeamento de espaço de dados abstratos multinível

Figura IV.3 : Mapeamento dos Espaços de Dados Abstratos

A figura IV.3 ( b ), ilustra um caso mais geral. Um programa $E_3$ utiliza objetos ( recuperáveis ou irrecuperáveis ) suportados por $I_0$ e pela extensão $E_2$, cujas representações concretas são respectivamente, $C_3$ e $A_2$. Porém, o programa extensão $E_2$, utiliza objetos disponíveis na extensão $E_1$ para implementação de seus objetos, que por sua vez usa somente objetos recuperáveis de $I_0$. ( $E_1$ poderia utilizar objetos irrecuperáveis, para os quais ele deve prover meios de suas restaurações; veja parágrafo anterior ).

Para esse caso, se $E_3$ for recuperado, $E_2$ seria sinalizado para realizar as ações de recuperação de seus objetos irrecuperáveis usados por $E_3$. Todavia, $E_2$ espera que todos os seus objetos recuperáveis

sejam automaticamente recuperados, embora alguns deles tenham sido implementados por $E_1$.

Assim, resumindo, quando um programa $E_i$ é recuperado, todas as extensões de mais baixo nível utilizadas, $E_1$, $E_2$. ..., $E_{i-1}$, serão solicitados a restaurarem os objetos abstratos por elas mantidos, para os seus estados anteriores. O interpretador subjacente $I_0$ deve garantir que isso ocorra, sinalizando todas as extensões relevantes até que todas as restaurações tenham sido completadas. Nesse ponto, o programa $E_i$ pode reiniciar a sua execução. Como no esquema anterior, o interpretador subjacente terá ( conceitualmente ) que sinalizar as extensões $E_1$, $E_2$, ..., $E_{i-1}$, sempre que o programa $E_i$ estabelecer ou descartar um ponto de recuperação.

Nesse esquema, deve-se tomar o cuidado para que os ambientes das partes de programas de recuperação de uma extensão sejam locais. A necessidade desse comportamento se deve ao seguinte: se as estruturas de dados usados para manter os dados de recuperação de uma extensão forem tomadas do ambiente de recuperação do programa que a utiliza, então qualquer recuperação deste, resultaria que aquelas estruturas de dados seriam automaticamente restauradas, apagando os dados de recuperação armazenados pela extensão. Isso pode ser evitado de dois modos, a saber:

a) garantindo que as partes de programas de recuperação da extensão ( ou o interpretador subjacente ), não utilizem quaisquer objetos recuperáveis que estão contidos pelo ambiente de recuperação do programa que a utiliza.
b) fazendo com que o interpretador subjacente seja capaz de distinguir programas e programas de recuperação de uma extensão e, assim, determinar quando aplicar as regras do esquema disjunto.

Diante dessa situação, parece apropriado que um interpretador básico suporte ambos os esquemas para uma implementação eficiente de recuperação em Sistemas Multiníveis.

### IV.1.2 ABORDAGEM PLANEJADA DE RECUPERAÇÃO PARA SISTEMAS DISTRIBUÍDOS

Até aqui, discutiu-se os métodos e técnicas para a construção de sistemas tolerantes a falhas ( principalmente, a falhas do tipo não previsíveis ), tendo-se em mente que tais sistemas são do tipo

centralizado. Isto é, não foram considerados os problemas decorrentes do fato do controle, dos programas e dos dados do sistema, agora, poderem estar distribuídos. A seguir, esses aspectos os serão, seguindo SHRIVASTAVA ( 1981 ).

Para Sistemas Distribuídos, a implementação do esquema disjunto de recuperação é bastante complexa pela simples razão de que, agora, os objetos recuperáveis utilizados por um processo podem estar localizados em nodos distintos. Conseqüentemente, o interpretador subjacente ao processo, não conterá as abstrações dos objetos remotos. Diante disso, o outro esquema se mostra mais adequado como esquema de recuperação para Sistemas Distribuídos.

Como visto, o esquema de recuperação disjunto, quando solicitado a executar a operação de recuperação de uma região devido a ocorrência de um erro, restaura todos os objetos recuperáveis e irrecuperáveis utilizados, por meio dos dados armazenados na correspondente região de dados para recuperação. Entretanto, agora, esses objetos podem não estarem localizados no mesmo nodo que o processo que os utiliza. Obviamente, a forma de tratamento desses objetos, deverá ser diferente daquele para objeto local, pois, executar a recuperação de um objeto remoto por meio de comunicação, no mínimo exigirá um tempo muito aquém do viável.

O método sugerido por SHRIVASTAVA ( 1981 ) para o tratamento de objetos remotos é o seguinte: para se acessar um objeto remoto, primeiro cria-se um operador no nodo onde se localiza tal objeto; em seguida envia-se mensagens para ele acessar o objeto. Para tal, o método exige que:

1) a operação de criar um operador seja recuperável; isto é, operadores sejam eliminados durante a recuperação e,
2) se são enviadas mensagens para acessar objetos recuperáveis, as suas recuperações sejam também comandadas por meio da emissão de mensagens para recuperação.

Para ilustrar o método, admita que um programa P sobre o controle de um processo Q, acesse um objeto Z, lotado em um outro nodo ( $N_i$ ); e que tal objeto tem sua representação concreta também distribuída - objeto recuperável A em algum nodo remoto ( $N_s$ ), objeto recuperável B no nodo local ( o mesmo do programa P ) e um objeto irrecuperável C em um outro nodo remoto ( $N_k$ ). Durante a execução de P por Q, um operador será criado no nodo $N_i$ ( $w_i$ ) e quando este inicia sua execução, serão criados os operadores para A ( $w_s$ ) e C

( $w_k$ ). Os operadores $w_s$ e $w_k$ serão destruídos quando a execução de $p_k$ terminar e $w_i$, quando $p_j$ terminar.

Admite-se a existência de um manipulador de objetos remotos por meio do qual cria-se ou elimina-se processo. Tal manipulador implementa as seguintes facilidades sob o comando de seus usuários: sempre que um processo estabelece ( ou descarta ) um ponto de recuperação, mensagens são enviadas aos operadores relevantes daquele processo; se um processo se recupera para o i-ésimo ponto de recuperação, mensagens são enviadas para os operadores relevantes e quaisquer operadores que foram criados nas regiões de recuperação eliminadas, são destruídos.

Em suas conclusões, Shrivastava salienta que a hipótese crucial feita foi a da necessidade de fazer com que as ações de recuperação de processos fossem independentes, isto é, que um processo não use um objeto que esteja em estado não confiável, o que pode degradar seriamente o desempenho. Cita, porém, algumas referências que procuram contornar o problema, como feita por Davies e Bjork's.

## IV.2 ABORDAGEM NÃO PLANEJADA DE RECUPERAÇÃO DE SISTEMAS DISTRIBUÍDOS

Essa abordagem para a recuperação de Sistema Distribuído, é bastante distinta da anterior. Naquela, os processos envolvidos se preocupam em coordenar os pontos de recuperação, como os seus estabelecimentos e também os seus comprometimentos ( descartes ), a fim de recuperar o sistema, diante da ocorrência de alguma falha. Por, justamente, os processos estarem preocupados com essas tarefas, o esquema acaba sacrificando a velocidade de processamento e impondo algumas restrições a respeito dos mecanismos de comunicação. A fim de retirá-las, a idéia dessa abordagem é liberar os processos dessas preocupações ( de coordenarem seus pontos de recuperação e seus comprometimentos ), não impondo, dessa forma, nenhuma estrutura sobre o sistema, permitindo que este tenha um alto grau de autonomia e identidade funcional. Fica a cargo de algum protocolo, ou um mecanismo externo ao sistema, o controle da recuperação do sistema, quando da detecção de algum erro por um processo que o ativa, fazendo com que ele determine um conjunto de pontos de recuperação para os quais os processos devem recuar, que represente um estado consistente do

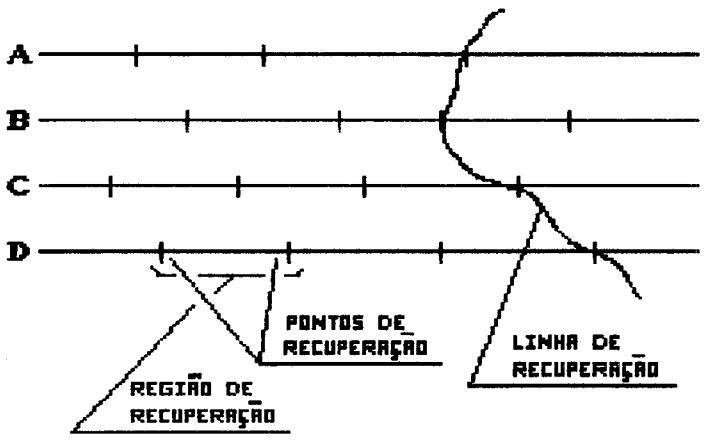



Figura IV.4: Pontos de Recuperação, Região de Recuperação e Linha de Recuperação

sistema.

Um estado consistente do sistema, nessa abordagem, é, então, composto por um conjunto de pontos de recuperação dos processos envolvidos, que MERLIN e RANDELL ( 1978 ) denominaram de linha de recuperação. A figura IV.4 ilustra esses conceitos ( WOOD, 1981 ).

A procura por uma linha de recuperação, na realidade, corresponde à procura de uma ação atômica ainda incompleta, que pode ser, assim, denominada de ação atômica não planejada.

Em Sistemas Distribuídos, a construção de tal linha é bastante complexa, pois durante esse processo, há a possibilidade da ocorrência de novas interações, justamente devido ao grau de funcionalidade dos processos que o esquema deseja oferecer.

MERLIN e RANDELL ( 1978 ) discutem esses aspectos e apresentam idéias e técnicas que, segundo eles, podem ser diretamente implementadas ou utilizadas como referências para a validação de outros mecanismos de recuperação de erros para trás, para Sistemas Concorrentes. Eles não chegam a definir um algoritmo, mas apresentam um protocolo para a construção de linhas de recuperação, o qual denominaram de "Protocolo Chase". Mostram as dificuldades para a execução dessa tarefa, principalmente, quando se pode seguramente descartar um ponto de recuperação de um processo, pois uma vez descartado um ponto de recuperação de um processo, dificilmente se pode recolocá-lo naquele estado posterior.

Preocupado com isso, WOOD ( 1981 ), descreve protocolos para a determinação de quando seguramente se pode descartar um ponto de recuperação de um processo e mostra a surpreendente complexidade desse problema. Os protocolos apresentados, o autor, denominou-os de

otimista e pessimista. Ambos considerando a possibilidade da necessidade de se ter o aninhamento de regiões de recuperação.

Basicamente, sem entrar em maiores detalhes, a estratégia de recuperação não planejada de Sistemas Distribuídos, devida a MERLIN e RANDELL ( 1978 ) e WOOD ( 1981 ), é a seguinte:

a) o esquema para a recuperação se baseia na história do sistema, ou seja, o fluxo de informação observado. As informações a serem retidas são aquelas relevantes como os pontos de recuperação de cada processo, como as trocas de mensagens ocorreram entre os processos, etc.;
b) para a retenção de tal fluxo de informação, cada nodo do Sistema Distribuído fica responsável pela parte que lhe toca, isto é, se responsabiliza em reter a história do fluxo de informação que ocorre nesse nodo;
c) quando um processo percebe a ocorrência de uma falha, o protocolo responsável pela procura de um estado consistente, é ativado;
d) o protocolo inicia a recuperação desse processo e por meio de emissão de mensagens de recuperação a todos os processos que se comunicaram com ele ( registrado no fluxo de informação observado ), desde o seu último ponto de recuperação até o instante da ciência da ocorrência de uma falha, procura ir colocando os processos envolvidos diretamente ou indiretamente em seus estados anteriores, que por sua vez influem outros da mesma forma, e assim por diante;
e) esse fluxo de mensagens de recuperação termina, quando não é mais necessário o envio de tal tipo de mensagem a mais nenhum processo.

Note, assim, que há a possibilidade de enquanto o processo de recuperação estiver em atividade, os processos do sistema estão trabalhando livremente, salvo aqueles que foram suspensos pelo protocolo, para fins de recuperação e ficaram a espera de um sinal para poder prosseguir. Note que, dessa forma, se sacrifica o mínimo necessário da velocidade de processamento do sistema, uma vez que os processos não envolvidos continuam funcionando normalmente. Pode, entretanto, ocorrer de algum processo envolvido continuar o processamento, quando seria interessante que ele interrompesse suas operações, pois haverá a necessidade de logo em seguida desfazer as operações realizadas. Isso se deve ao tempo que decorre entre o instante que é iniciado a transmissão das mensagens pelo protocolo até

chegar a todos os seus destinos. Este é um dos custos dessa estratégia que percebe-se que é bem menor do que parar todo o sistema para a sua recuperação.

Ambas as abordagens para recuperação de Sistemas Distribuídos, utilizam a técnica de rolar o sistema para trás. Elas são apropriadas para diferentes aplicações. A abordagem planejada é adequada para ser aplicada em ambientes para processamento de transações, diante das características desse tipo de sistema, enquanto que a abordagem não planejada é mais apropriada onde os processos tenham um alto grau de autonomia e identidade funcional; por exemplo, em ambiente de controle de processos distribuídos. Naturalmente, existem outras estratégias intermediárias, cada uma procurando, mediante as características do sistema, simplificar o esquema do mecanismo de recuperação a ser implementado.

Alguns autores reportaram seus esforços de projeto e implementação de mecanismos de recuperação de sistemas concorrentes. Por exemplo, JEGADO ( 1983 ), apresenta as características de recuperabilidade de Sistemas de Arquivos Distribuídos; BROWNBRIDGE et alii ( 1982 ), apresenta os experimentos de mecanismos relativos a segurança e confiabilidade no sistema "UNIXes of the World Unite"; etc.

## IV.3 MECANISMOS PARA RECUPERAÇÃO DE PROCESSO

Nos dois itens anteriores, foram discutidas duas abordagens que procuraram descrever métodos genéricos para recuperação de Sistemas Distribuídos, quando da detecção de um erro que ocorreu devido a uma falha não prevista ou de falhas cujas formas de tratamento devem ser por meio de desfazer operações realizadas. Elas não colocaram nenhuma restrição quanto ao tipo de sistema de troca de mensagens utilizado, como também, nada explicitaram se existiria algum cuidado especial se a recuperação de um processo residente em um nodo operador, devesse ser feita em um outro nodo, nada mencionaram sobre o tipo do meio físico de comunicação que poderia ser usado, etc. Enfim, a diretriz dessas abordagens foi a de, na ocorrência de um evento indesejável daquele tipo, descrever um método que coloque o Sistema Distribuído em um estado consistente que já existiu, ou poderia ter existido; diante do assincronismo natural do sistema. Para tanto, as suas preocupações se fixaram em como determinar uma linha de recuperação consistente,

para onde os processos diretamente e indiretamente afetados por algum erro deveriam retroceder, para que o sistema recuasse para um estado consistente e a partir daí pudesse prosseguir normalmente com a execução de suas funções, sem que isso causasse uma queda inadmissível ao seu desempenho. É devido a essa generalidade, que as soluções mencionadas apresentam uma complexidade natural.

Agora a preocupação consiste em tratar algumas especificidades dos vários eventos indesejáveis que podem surgir em um processo de um Sistema Distribuído, e que podem fazer com que o seu comportamento seja desviado de sua especificação. Além disso, continuam sendo tratados aqueles provenientes de defeitos de "hardware" que ocasionam o colapso de um processo ( deterioração ou impossibilidade de acessar o código de um processo ), ou seja, eventos como: defeito de parte de memória onde reside o código de um processo e impossibilidade de acessar um nodo operador, como colapso do nodo operador ou colapso de nodo de comunicação ou rompimento de linhas de comunicação. Tais tipos de eventos indesejáveis podem ser somente tratados dinamicamente, a fim de evitar que o sistema entre em colapso, se existir redundância dos processos não mais acessíveis, em algum outro nodo operador.

Para Sistemas Distribuídos, a tolerância a falhas é um de seus objetivos para a construção de sistemas altamente disponíveis. A idéia é utilizar os vários computadores componentes do sistema, interligados por algum subsistema de comunicação, como elementos redundantes ativos ( redundância natural ). Caso algum deles entre em colapso, um outro pode assumir suas funções também. O modelo de usuários-servidores de Sistema Distribuído apresenta o problema da falta de redundância natural dos processadores servidores. Já o outro modelo, o simétrico, apresentará o mesmo problema, caso não exista um dispositivo periférico equivalente ao compartilhado, que a princípio estava sendo utilizado de forma particular a um determinado processador. De qualquer modo, para ambos os modelos deve-se introduzir mecanismos para tolerar falhas daqueles tipos, salvaguardadas as impossibilidades citadas.

A seguir, levantam-se os problemas que devem ser consideradas pelos mecanismos de recuperação de processos.

## IV.3.1 QUESTÕES SOBRE RECUPERAÇÃO DE PROCESSOS

Várias são as questões que influenciam a definição de um mecanismo de recuperação de processos para Sistemas Distribuídos. A existência ou não de dispositivos especiais como um relógio global e de memória de acesso rápido e estável para armazenamento de cópias de segurança ( "back-ups" ). Além disso, influenciam: o subsistema de comunicação ( o tipo da rede de comunicação - barramento ou ponto a ponto, a topologia da rede - conexão ponto a ponto, em anel, estrela, hierárquica, etc. - o sistema de troca de mensagens a nível de nodos de comunicação ou nodos operadores ); o sistema de troca de mensagens a nível de processos; etc.

Dessas, se levará em consideração, as seguintes:

- existência ou não memória de acesso rápido e estável;
- tipos de processos: processos de usuários e processos do Sistema Operacional
- sistema de troca de mensagens: a nível de processos e a nível de subsistema de comunicação: rede de comunicação: barramento e ponto a ponto com os seguintes tipos de topologias: em anel e conexão ponto a ponto. A rede pode ser confiável no sentido da existência de redundância, controlada por meio de circuitos elétricos eletrônicos; sistema de troca de mensagens a nível de nodos de comunicação ou nodos operadores;

Quanto ao relógio global, ele tem sido proposto por analogia às propostas feitas para recuperação de processos em Sistemas Centralizados, o qual facilitaria a sincronização para o salvamento dos estados dos processos. Entretanto, tais tipos de propostas, de um modo geral, obriga que todo o sistema pare pelo tempo necessário para a execução de salvamento de tal ponto para recuperação do sistema, degradando o desempenho. Assim, é interessante a elaboração de uma que não exija um único instante para salvar um ponto de verificação do sistema.

### IV.3.1.1 EXISTÊNCIA OU NÃO DE MEMÓRIA DE ACESSO RÁPIDO E ESTÁVEL

A existência de memória de acesso rápido e estável tem a sua importância para o armazenamento das cópias de segurança ( "back-

ups" ) dos processos que deverão ser recuperados na ocasião das perdas de seus originais. A característica de ser de acesso rápido é por questão de desempenho; lembre-se que a recuperação é um serviço do sistema que visa deixá-lo disponível aos usuários o maior tempo possível dentro de um determinado intervalo, assim, ela não é vista por estes como algo produtivo. A estabilidade é por questões óbvias, pois, se ela não for, a queda de energia ou uma oscilação mais brusca, comprometeria as cópias de segurança e com isso seus objetivos.

A localização ( ou localizações, no caso de se utilizar mais do que uma ) desse tipo de memória no Sistema Distribuído, é muito importante para esses fins. Por exemplo, caso ela fique conectada diretamente a um nodo operador cuja segurança contra defeitos é igual aos demais nodos do Sistema Distribuído, na ocorrência de uma falta nesse nodo que implique em seu colapso, essa memória se tornará acessível somente quando tal nodo for consertado e devidamente ativado. Tal espera, provavelmente já terá comprometido os objetivos da utilização dessa memória no Sistema Distribuído. Assim, poderia-se colocar em um ou mais nodos especiais que apresentam uma segurança bem maior que os demais. Não se utiliza tais nodos para todo o Sistema Distribuído por questão de custo, salvo exceções. É uma solução barata utilizar o dispositivo de disco magnético - memória secundária ( "Winchester", por exemplo ) como memória para esses fins. POWELL e PRESOTTO ( 1983 ), utilizam tal tipo de esquema; um computador superprotegido com uma memória secundária de alta capacidade para fins específicos de recuperação de processos do Sistemas Operacional DEMOS/MP ( a ser visto no item IV.3.2 ). Um alternativa seria a colocação de tal dispositivo de memória secundária acessível por mais do que um nodo operador, entretanto, apenas um deles tendo o direito até que esse

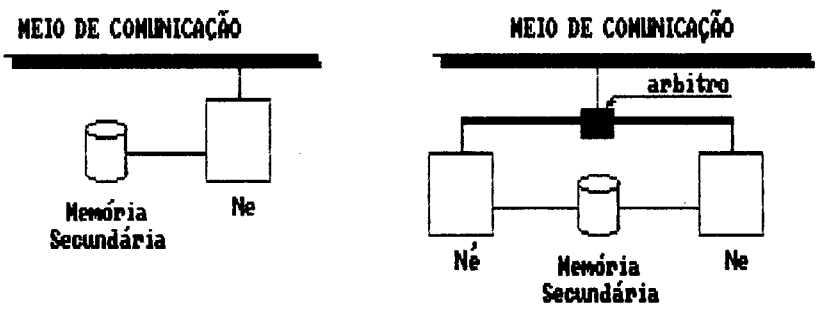



Figura IV.5: Esquemas de colocação de dispositivos de memória para fins de recuperação de processos

seja explicitamente passado para o outro. A figura IV.5 ilustra esses esquemas.

Na ausência ou não utilização desse tipo de memória para fins de recuperação de processos, uma alternativa é aquela proposta por BORG et alii ( 1983 ) - a ser vista no item IV.3.2. Nessa, o esquema é aplicado no sistema Auragen, onde existe uma redundância de vias de comunicação entre grupo ( "cluster" ) de processadores ( cada um com fins específicos ) e com os dispositivos periféricos sendo acessíveis por dois "clusters". O esquema de um modo geral é o seguinte: para cada processo residente em um "cluster", existe uma cópia em um outro "cluster"; quando o principal é perdido, a cópia se torna o original e é criada uma nova cópia.

### IV.3.1.2 APLICAÇÕES DE USUÁRIOS

Recuperar aplicação de usuário tem sentido se ela for uma aplicação distribuída e portanto um Sistema Operacional Distribuído para fins específicos, ou for composta de um único programa cujo tempo total de processamento é muito grande. Para o primeiro caso, a importância da tolerância a colapso de processadores é direta. Quanto ao segundo caso, admitindo que o programa seja totalmente seqüencial, é admissível que na ocorrência de colapso do processador que o estava executando, ele possa ter o seu processamento continuado em um outro processador disponível no Sistema Distribuído. Isso, desde que, na média, o tempo gasto para recuperação seja menor do que iniciar novamente a execução do programa.

Entretanto, como a existência de um mecanismo de recuperação de processos é importante em Sistemas Distribuídos, ele pode ser construído para fazer ou não a distinção entre processos de usuários e do sistema. Caso não faça, então a recuperação de aplicações de usuários deverá ser automática. Caso contrário, o Sistema Operacional coloca ou não à disposição dos usuários, primitivas ou diretivas ( comandos para o Sistema Operacional ) para que eles possam se utilizar dessa facilidade. Essa opção deverá ser pesada em termos de custo adicional de "software", de memória e de dispositivos especiais.

### IV.3.1.3 SISTEMA DE TROCA DE MENSAGENS

Uma preocupação levantada nos estudos descritos no item IV.1, foi a da determinação de uma linha de recuperação consistente, ou seja, estados dos processos salvos, em que não houve nenhuma troca de mensagem entre um processo após essa linha com um outro processo antes dessa linha. Em termos ilustrativos uma linha de recuperação é consistente se ela não corta nenhuma troca de mensagens entre processos ( veja a figura IV.4 ). Dessa forma, o esquema de recuperação procura colocar todos os processos afetados pela recuperação de um ou mais processos devida à falha, começando por estes, em uma verdadeira operação de desmonte. Uma vez os processos colocados nesses estados consistentes, eles podem dar prosseguimento aos seus processamentos. Está garantido que o sistema foi colocado em um estado anterior à ocorrência do evento indesejável e desfeitas todas as operações realizadas ( incluindo-se as interações - troca de mensagens ) a partir deste e, assim, como se os estados posteriores àquele, nunca tivessem existidos. Dessa forma, note que, o processamento que segue terá seu rumo determinado pelos próprios processos constituintes, como se nada tivesse acontecido.

É visível que um esquema desse tipo funciona adequadamente se a recuperação dos processos de um sistema seja feita com ele colocado em um estado de espera pelo término dessa tarefa. Entretanto, no item IV.2, foi observado que essa forma é muito restritiva e seria interessante, em termos de desempenho, dar liberdade aos processos que não foram atingidos pela falha de prosseguirem com os seus processamentos. Isso, porém, pode gerar vários problemas, a saber alguns:

- um processo pode enviar mensagem a um processo que está em recuperação, a qual será feita em algum outro processador ( por exemplo, um processo servidor, cujo dispositivo entrou em pane e que existe um outro equivalente em um outro nodo operador ), assim, a mensagem enviada poderá não chegar ao seu destino, ficando esta órfã podendo gerar graves problemas, a menos que o sistema de troca de mensagens resolva contornar devidamente essa ocorrência indesejável.
- um processo deseja receber de forma bloqueada uma mensagem de um processo que está sendo recuperado: dependendo do sistema de troca de mensagens e de funções de supervisão do núcleo, podem

surgir vários inconvenientes. Por exemplo, caso exista um temporizador de espera e este ultrapasse o limite estabelecido ( percebido pelo núcleo ) antes que o processo emissor seja devidamente recuperado, e difundido o seu novo endereço e envie a mensagem esperada, o núcleo pode querer eliminar o processo receptor.

- um processo pode querer receber uma mensagem de forma não determinística que por coincidência, todos os seus emissores estão em recuperação: isso pode causar que o processo consumidor levante uma exceção cuja forma de tratamento seja o aborto do processo, ou um tratamento envolvendo medidas que devam ser tomadas pelo núcleo, como aborto do sistema "dono" do processo, ou etc.

Percebe-se, então que, o sistema de troca de mensagens, a ser utilizada por sistemas que visam a recuperação dinâmica de processos cooperantes, deve levar em consideração os esquemas adotados para tal. Em outras palavras, se quer dizer que é interessante que o sistema de troca de mensagens e o sistema de recuperação de processos funcionem em concordância um com o outro.

Apenas para ilustrar uma "solução" para os inconvenientes acima, o sistema de recuperação poderia suspender a execução de todos os processos do sistema a ser recuperado o tempo suficiente para marcar os processos que deverão ser recuperados e definir seus novos endereços ( atualizando todas as tabelas necessárias ), para depois então começar a executar a tarefa da recuperação propriamente dita, ao mesmo tempo em que é liberada a execução em concorrência dos processos não afetados. Ao término de cada processo recuperado este é desmarcado e colocado como pronto para ser executado normalmente. Dessa forma, o sistema de troca de mensagens, pode tomar atitudes adequadas para os problemas citados. Por exemplo:

- quando um processo não marcado como em recuperação, enviar uma mensagem de forma bloqueante para um marcado, ele pode ficar suspenso até que o destinatário esteja em ordem, ignorando o temporizador, ou acrescentado-o de um tempo estimado para tal.
- qualquer processo que queira receber uma mensagem de forma bloqueante de um que está em recuperação, pode ficar suspenso de forma análoga ao caso anterior.

Como essa solução teve apenas o caráter ilustrativo, foi implicitamente admitido que os processos não marcados para serem

recuperados, não serão afetados por aqueles durante a fase de recuperação, no sentido de precisarem posteriormente desfazerem as operações realizadas, como também, a detecção dos eventos e seus tipos que exigiram que determinados processos tiveram que serem recuperados. A intenção foi a de reforçar a afirmativa da influência do sistema de troca de mensagens sobre as técnicas de recuperação de processos.

Além disso, para que o sistema de recuperação de processos possa executar as tarefas preliminares, do exemplo acima, ele necessita enviar mensagens para os nodos operadores que contém os processos que devem ser recuperados, para que estes marquem-os como tais, como também, para a atualização das tabelas, salvamento dos pontos de verificação e etc. Assim, nota-se que, tais tipos de mensagens devem ser enviadas e atendidas no menor tempo possível por questões de desempenho, tanto para a construção de linhas de recuperação consistentes, como para evitar uma maior propagação dos erros e conseqüentemente diminuindo a tarefa de desfazer operações. Dessa forma, percebe-se que o tipo da rede de comunicação e sua topologia acaba influindo na esquematização daquele sistema, como se descreverá a seguir.

Os tipos de redes de comunicação a serem consideradas nesta análise são a baseada em barramento e conexão ponto a ponto. Essa característica do Sistema Distribuído influi em um esquema de recuperação de processos, principalmente, quanto à questão de desempenho, a qual pode simplificá-lo, pelos seguintes motivos:

- para rede de comunicação baseado em barramento: todas as mensagens que devem ser enviadas pelo sistema de recuperação de processos podem ser feitas por meio de difusão e sabe-se que elas serão ouvidas "simultaneamente" por todos os nodos operadores. Por exemplo, a mensagem para o salvamento dos estados dos processos para a constituição de um ponto de verificação do sistema pode ser atendida por todos os nodos operadores quase que instantaneamente. Obtendo-se, assim, praticamente, o estado completo do sistema em um determinado instante ( a diferença do tempo entre o salvamento do estado de um processo em relação a um outro residente no mesmo nodo processador é "igual" ao que está localizado em um nodo distinto );
- para rede de comunicação baseado em conexão ponto a ponto: para esse tipo de rede, a conexão pode ser feita em laço, estrela, árvore,

completa ou de forma irregular ( KIRNER, 1986 e KIRNER e MENDES, 1988 ). Dessas topologias considerar-se-á para fins ilustrativos, apenas as em laço e conexão completa. As mensagens difundidas pelo sistema de recuperação de processos ( o qual está localizado em algum nodo da rede ), tanto em uma ou outra topologia considerada, não serão ouvidas ao mesmo tempo. A defasagem entre os tempos do salvamento do estado dos processos residentes em um nodo em relação aos localizados em outros nodos é razoavelmente grande em relação ao tipo de rede anterior. Primeiro porque, a difusão de mensagens nesse, pode ser feita pelo envio individual para cada nodo ( conexão completa ) ou pela solicitação de sua retransmissão do nodo receptor para o seguinte ( em laço ). Depois porque, sempre existirá mais do que uma disputa para o acesso ao meio de comunicação. Além disso, o protocolo de comunicação, por questões de controle de fluxo de informações na rede, pode se utilizar de "buffers", o que poderá tornar a defasagem maior ainda.

Caso o sistema de troca de mensagens resolva devidamente os inconvenientes acima explicitados e outros não mencionados, ainda resta o problema da construção de linhas de recuperação consistentes, a qual ainda não se tem proposta cuja complexidade seja passível de implementação a um custo viável. Diante disso, se mostram interessantes as propostas a serem analisadas a seguir, devidas a BORG et alii ( 1983 ) e POWELL e PRESOTTO ( 1983 ), que evitam a determinação de linhas de recuperação consistentes, para a recuperação de processos em Sistemas Distribuídos, baseando-se essencialmente no sistema de troca de mensagens.

## IV.3.2 PROPOSTAS DE MECANISMOS PARA RECUPERAÇÃO DE PROCESSOS BASEADO NO SISTEMA DE TROCA DE MENSAGENS

A seguir discutem-se duas propostas de mecanismos para recuperação de processos que diferem substancialmente das anteriores no sentido de que, nestes não há a preocupação com as linhas de recuperação, entretanto, impõem uma restrição, de que os processos devem ser determinísticos. Essas propostas são devidas a BORG et alii ( 1983 ) e POWELL e PRESOTTO ( 1983 ).

BORG et alii ( 1983 ) se basearam no requisito de que os processos devem ser determinísticos; ou seja, se dois processos se iniciam a partir de estados idênticos e recebem entradas idênticas, eles serão executados identicamente e, conseqüentemente, produzirão resultados idênticos. Com isso, para a recuperação de um processo afetado pela ocorrência de uma falha, é necessário apenas parar explicitamente esse processo, colocá-lo em um estado anteriormente salvo ( um ponto de verificação ) e apresentar a ele as mesmas entradas e na mesma seqüência. Para tanto, eles apresentaram um sistema de mensagem que sempre envia atomicamente a mensagem para três destinos: para o destino propriamente dito e para as cópias do destino e emissor. Cada cópia, preferencialmente, residente em um nodo operador distinto de seu principal. Para o destino principal, a mensagem é recebida segundo a semântica definida para os usuários pelo sistema de troca de mensagens. Para o destino cópia, a mensagem deve apenas ser armazenada para fins de recuperação; para não perder as mensagens já enviadas porém ainda não efetivamente recebidas, desde o último ponto de verificação, quando da recuperação do processo destino. Já para o emissor cópia, a chegada da mensagem serve apenas para contabilizar a quantidade de mensagens enviadas desde o último ponto de verificação; para evitar o envio repetido de mensagens quando da recuperação do processo emissor. Como cada processo pode ser tanto emissor como receptor, sua cópia possuirá uma fila de mensagens que foram enviadas para o seu principal e um contador da quantidade de mensagens enviadas pelo seu principal. Além disso, o processo principal deve manter a quantidade de mensagens já recebidas desde o último ponto de verificação. A atualização dessas informações, feita a cada estabelecimento de ponto de verificação, é efetivada por meio de uma mensagem de sincronização que contém uma pequena quantidade de informação de estado, enviada diretamente para o núcleo do nodo operador que contém a cópia do processo, cujo ponto de verificação está sendo estabelecido; uma informação contida nessa mensagem é o contador de mensagens consumidas desde o último ponto de verificação. Tal mensagem de sincronização, permite que as mensagens retidas para o processo cópia possam descartar aquelas que o seu principal já consumiu ( pelo contador de mensagens consumidas ) e zerar o seu contador de mensagens enviadas. Quanto à frequência de estabelecimento de ponto de verificação, pode ser pela quantidade de mensagens enviadas e recebidas, ou por intervalo de tempo ou por

algum outro método. Observa-se porém que, isso carece de uma boa análise levando em consideração, o tempo de processamento utilizado para o salvamento do estado do processo, o meio físico de comunicação, a quantidade de memória para reter as mensagens da cópia do destino, etc. Nesse esquema, a recuperação de um processo é direto. O núcleo do nodo operador que contém a cópia do processo que entrou em colapso, ao receber tal informação, torna a cópia o novo principal e cria uma nova cópia.

Já POWELL e PRESOTTO ( 1983 ), atribuem essas tarefas de gerenciamento, armazenamento das cópias e responsabilidade da recuperação a um único nodo operador superprotegido ( controle centralizado - veja figura IV.6 ), denominado de nodo de recuperação, e pressupõem que os processos estejam preparados para desfazer as operações realizadas por transações incompletas ( uma transação sempre leva o sistema de um estado consistente para um outro ). Segundo eles, enquanto que naquele esquema se utiliza tempos e memórias dos processadores para a execução de aplicações, para salvar as informações redundantes para serem usadas para as recuperações, quando do surgimento do evento de um colapso de um processo ou processador, nesse, tudo isso fica centralizado em um único nodo operador especificamente colocado para esses fins, liberando os processadores para as aplicações daqueles inconvenientes. Além disso, consideram que esse esquema deve ser mais confiável, pois o único ponto fraco, excluindo o meio físico de comunicação, o qual vale para o outro esquema, é esse nodo operador. Daí, a sugestão dele ser superprotegido, se justifica o custo. Esse esquema, por não permitir que as cópias se tornem seus principais, como no de BORG et alii ( 1983 ), a recuperação de um processo é um tanto diferente. Existe um processo gerente residente no nodo de recuperação, responsável pela tarefa de recuperação de um processo que, quando notificado da ocorrência de tal evento, executa os seguintes passos:

1) Definir o nodo operador onde deverá ser recuperado o processo. O nodo pode ser o mesmo se isso for possível. Geralmente, a falha é proveniente da apresentação de falta em nodo operador. Assim, o processo a ser recuperado, normalmente, ocorrerá em um outro nodo operador ( uma espécie de migração ).
2) Enviar uma mensagem ao núcleo do nodo operador destino, solicitando que ele inicie um processo com a identificação do processo especificado e coloque-o para o estado de em recuperação.

Transmitir a informação do último ponto de verificação para permitir que o núcleo o regenere para aquele instante do ponto de verificação. Também notificar o núcleo de quando parar de ignorar as mensagens enviadas por esse processo, determinado pelo número de mensagens já enviadas e recebidas pelo processo destino. O processo pode então reassumir o processamento normal.

3) Enviar para o processo em recuperação, todas as mensagens que ele tinha recebido entre o instante de seu último ponto de verificação até o instante do seu colapso.

Note que esses esquemas recuperam processos diretamente atingidos pela ocorrência de uma falha, colocando-os, primeiramente, aos estados anteriores a este evento e mascaram-no por meio das reexecuções apresentando a eles as mesmas mensagens recebidas até o instante da detecção da falha. Dessa forma, tais esquemas induzem que um processo recuperado, tenha o mesmo comportamento até o instante da detecção da falha. Diferindo daqueles que procuram apenas colocar um processo a ser recuperado a um estado pertinente a uma linha de recuperação consistente e, a partir daí, o comportamento ficando por conta de si próprio. Observa-se, assim que, enquanto que essa última forma oferece a liberdade aos processos recuperados a prosseguirem

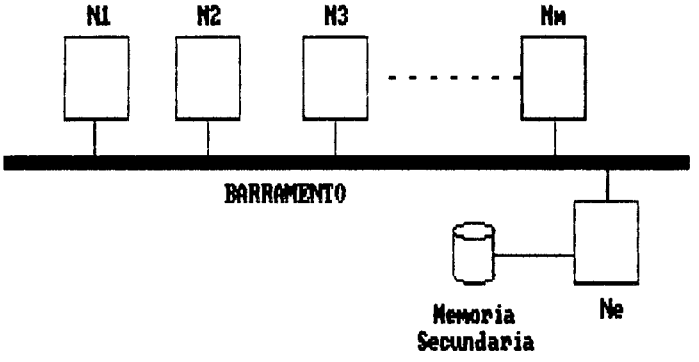



Figura IV.6: Arquitetura do Sistema Distribuído segundo a proposta de POWELL e PRESOTTO ( 1983 )

seus processamentos, como se nada de anormal tivesse ocorrido, naquelas, são impostas restrições até o instante da ocorrência da falha. Percebe-se que isso pode influir em processos que estão interessados em receber apenas as últimas informações. Por exemplo, controle de temperatura. Para contornar isso, basta apenas que o gerente de recuperação tome esse tipo de cuidado. Um outro tipo de problema que pode vir à mente seria quanto às operações não idempotentes. Porém,

POWELL e PRESOTTO ( 1983 ) já consideraram isso, quando disseram que os processos devem estar preparados para desfazer operações.

Concluindo a análise, POWELL e PRESOTTO ( 1983 ), na época, não tinham implementado os pontos de verificação, assim, um processo em recuperação, era na verdade reiniciado. Observaram, também que, não se preocuparam com as questões de: gerenciamento de memória, confiabilidade do registrador para recuperação, protocolos para registradores replicados, mecanismos para melhorar o desempenho das mensagens intra-nodos e tratamento de todos os processos em uma única máquina, como se fosse um único. Além disso, não abordaram os seguintes problemas, que se julgam importantes:

- as mensagens enviadas pelo recuperador de processos podem não chegar aos seus destinos;
- durante a recuperação, pode ser que o processo em recuperação ( ou o nodo operador que o contém ), entre em colapso.

## IV.4 MIGRAÇÃO DE PROCESSOS

Migração de processo é definido como a transferência de uma quantidade suficiente do estado de um processo de uma máquina para uma outra, de forma que ele possa ser ali continuado a sua execução, como se nada disso tivesse ocorrido ( SMITH, 1988 ).

A existência de mecanismo para execução de tal tarefa é interessante em Sistemas Distribuídos, permitindo, principalmente, melhorar o desempenho, para servir como uma ferramenta para o balanceamento dinâmico de carga, aumentar a sua confiabilidade ( ser tolerante a faltas ) e possibilitar o crescimento incremental, tanto em "hardware" como em "software" ( SMITH, 1988, KRAMER e MAGEE, 1985 ).

Para sistemas fortemente acoplados ( com compartilhamento de memória ), é notável que a construção de um mecanismo para a migração de processos é relativamente trivial e seria somente necessário caso ocorresse a falha ( ou melhor, falta ) de parte da memória, o que garantiria uma melhor confiabilidade, entretanto, em nada melhoraria o desempenho. Dessa forma, o estudo sobre a construção de tal tipo de mecanismo, se restringirá para Sistemas Distribuídos.

A migração de processos é desejável em Sistemas Distribuídos pelas várias razões acima citadas, que a seguir serão melhor detalhadas:

- para encontrar tolerância a certos tipos de faltas como: colapso de processo ( por falta em componentes de memoria ); colapso de nodo processador ou desligamento inadvertido de algum nodo pelo usuário local ou colapso de nodo de comunicação; colapso de um periférico integrante do sistema e que exista um substituto; etc.
- para melhorar a carga de cada processador, movimentando os processos entre eles ( balanceamento de carga ); isto pode melhorar o desempenho.
- para se ter um melhor desempenho, por exemplo, em sistemas de base de dados, quando um processo realiza uma redução de dados sobre um volume de dados maior do que o tamanho do processo, é mais vantajoso mover o processo para os dados ( SMITH, 1988 ).
- para atender características específicas de dispositivos que não podem ser acessadas remotamente, ou são inviáveis de serem assim feitas, por questões de tempo real ( SMITH, 1988 ).
- para permitir a construção de sistemas que podem crescer ou diminuir dinamicamente ( KRAMER e MAGEE, 1985 ).
- para permitir a construção de sistemas que não podem parar ou não é economicamente viável, para permitir a modificação de parte de seu "hardware e/ou de seu "software" ( KRAMER e MAGEE, 1985 ).

São vários os problemas a serem considerados para a construção de mecanismos para migração de processos para Sistemas Distribuídos, que a torna muito mais complexa do que para sistemas fortemente acoplados, a saber alguns:

a) os nodos operadores podem ser heterogêneos. Isso exigirá a tradução do código do processo ( a descrição de uma computação que pode ser executada por um computador ) e a necessidade de que o núcleo do nodo operador para o qual o processo migra, ofereça todas as primitivas utilizadas pelo processo naquele núcleo do nodo operador origem. Além disso, existe o problema da forma de representação de dados que devem ser compatibilizados de modo a não permitir inferências indesejáveis por parte dos nodos operadores receptores - MAGUIRE e SMITH ( 1988 ), analisa a migração de processos que realizam computações científicas e sugere como solução, a construção de uma representação externa única para todo o sistema.

b) independentemente se os nodos operadores são ou não homogêneos, a própria migração de processos é um gargalo para a produtividade do sistema. ZAYAS ( 1987 ) faz uma análise sobre esse problema, considerado por ele crucial, que trata essencialmente do custo da transferência do código do processo e de seu estado ( a imagem do processo ) de um nodo operador para um outro. Observa ele, que esse custo cresce linearmente com tamanho da imagem do processo a ser migrado, se a transferência for feita por cópia direta, ou seja, enquanto toda a imagem não for devidamente copiada no nodo destino, a sua execução está suspensa. Como uma solução para diminuir esse crescimento linear, e assim, sugerindo como uma técnica que pode ser adotada para se implementar mecanismos para migração de processos, ZAYAS ( 1987 ) demonstrou por meio de testes, que a cópia por referência é uma técnica bastante eficaz.

Isso não considerando que processos componentes de um sistema cooperam e assim, a migração de qualquer um deles gera vários problemas que devem ser solucionados em tempo de execução, diante da necessidade do restabelecimento das comunicações. Obviamente, tais problemas poderão ter soluções de pouca ou muita complexidade, dependendo da arquitetura do sistema e de outras características, tais como tipo de mecanismo de sincronização e comunicação utilizado, confiabilidade, etc.

A fim de melhor esclarecer, exemplifica-se. Em um sistema não distribuído e que seus processos se comunicam por meio de compartilhamento de memória através de monitores ( HOARE, 1974, HANSEN, 1978 ), havendo a necessidade de se migrar um de seus processos por motivo de tolerância a falta ( no caso, de parte da memória do computador ), o mecanismo para executar tal tarefa deverá se preocupar com os seguintes problemas:

- se o processo migrante contém pelo menos um monitor, haverá a necessidade de ajustar as chamadas das entradas do monitor de todos os processos que o utilizam.
- se o processo migrante não contém nenhum monitor, a tarefa do mecanismo para migração de processos é relativamente bastante simples, pois se resumirá em apenas realocar o código do processo e restabelecer o seu estado.

No primeiro caso, note que para solucionar o problema, há a necessidade de se conhecer todos os endereços dos pontos de cada processo onde é feita uma chamada de uma entrada do monitor. Para

isso, o ambiente de desenvolvimento de sistemas deve registrar essas informações ou o monitor deve ser especificado com uma lista de capacidade contendo além da identificação de seus usuários, os pontos onde ocorrerão as chamadas, ou um outro meio qualquer, desde que se possa precisamente identificar as posições de memória que contém endereço de uma entrada do monitor. Um problema nada trivial, que se deve ao tipo de mecanismo utilizado para a comunicação. Caso, o mecanismo utilizado fosse do tipo baseado em troca de mensagens, o problema a primeira vista poderia ser amenizado. Maiores detalhes será visto no item V.1.3 do capítulo seguinte.

De qualquer forma, dá para perceber que mecanismos baseados em compartilhamento de memória são mecanismos que "conectam" os processos cooperantes de forma muito rígida, espelhando bem a arquitetura de "hardware" de sistemas fortemente acoplados, exigindo assim, muito esforço para torná-los flexíveis o suficiente para permitir a migração de qualquer um de seus processos, a fim de encontrar uma certa confiabilidade, como já mencionado. Assim, os mecanismos baseado em troca de mensagens, que por não exigirem o compartilhamento de memória, deverão ser utilizados, pelo menos, se se deseja implementar mecanismos para a migração de processos. Cabe observar ainda que, mecanismos que não exijam o explicitamento do processo destino ( comunicação indireta ), facilitarão em muito a implementação da tarefa em pauta, pois a princípio, não haveria a necessidade de se alterar em nada o código de qualquer processo, quando da ocorrência da migração de um processo, componente de um sistema ( note o exemplo acima ).

Ainda em termos de mecanismos de comunicação, entretanto, para Sistemas Distribuídos e portanto baseados em troca de mensagens, existem as seguintes preocupações:

- pode ocorrer que, quando um processo está migrando, existam processos que enviaram mensagens para ele em seu endereço antigo ( mensagens em trânsito );
- será que as temporizações previstas para os envios síncronos de mensagens, admitidas corretas, continuariam adequadas?.

Tais preocupações deverão ser levadas em consideração quando da definição dos tipos de mecanismos para comunicação entre processos a serem utilizados, pois, como visto nos capítulos anteriores, é melhor tratar uma falha logo que ela é detectada a fim de se evitar a propagação de seus efeitos.

Preocupada com a questão referente ao problema das mensagens em trânsito, quando da execução da tarefa de migração, POWELL e MILLER ( 1983 ) - Migração de Processos em DEMOS/MP, propõem um mecanismo, que a seguir é descrito de forma resumida.

Essencialmente, o mecanismo foi elaborado pensando em apenas permitir o balanceamento dinâmico de cargas, em Sistemas Distribuídos compostos de computadores homogêneos. Assim, uma vez determinado que um específico processo deve ser migrado de um nodo operador para um outro, o responsável pela distribuição da carga no Sistema Distribuído, deve-se executar os seguintes passos:

1. Remover o processo da execução: o processo é marcado como em migração, para impedir que ele continue a receber mensagens;
2. Solicitar ao núcleo do nodo operador destino para mover o processo: enviar uma mensagem para o núcleo do nodo operador destino, solicitando que ele migre o processo;
3. Alocar um estado de processo no nodo operador destino: criar um estado de processo vazio no nodo operador destino, incluindo a reserva de espaço para o código do processo;
4. Transferir o estado do processo: por meio da facilidade de mover dados, o núcleo do nodo operador destino, copia o estado do processo em migração, para o estado de processo vazio.
5. Transferir o programa ( código ): por meio da facilidade de mover dados, o núcleo do nodo operador destino copia a memória ( código, dados e a pilha ) do processo para a memória do nodo destino.
6. Enviar as mensagens pendentes: após a notificação de que o processo está estabelecido no novo nodo operador, o núcleo fonte reenviar todas as mensagens que já estavam enfileiradas, ou que chegaram, desde que se iniciou a migração do processo;
7. Limpar o estado do processo: remover do nodo operador fonte, toda a informação a respeito do processo migrado. É deixado nesse nodo, um roteador de mensagens, para que ele possa enviar as mensagens que chegam para o processo migrado, para o seu novo endereço ( figura IV.7a ) e enviar o novo endereço para o processo emissor ( figura IV.7b );
8. Reiniciar o processo: o processo é reiniciado naquele estado em que estava quando se iniciou a migração.

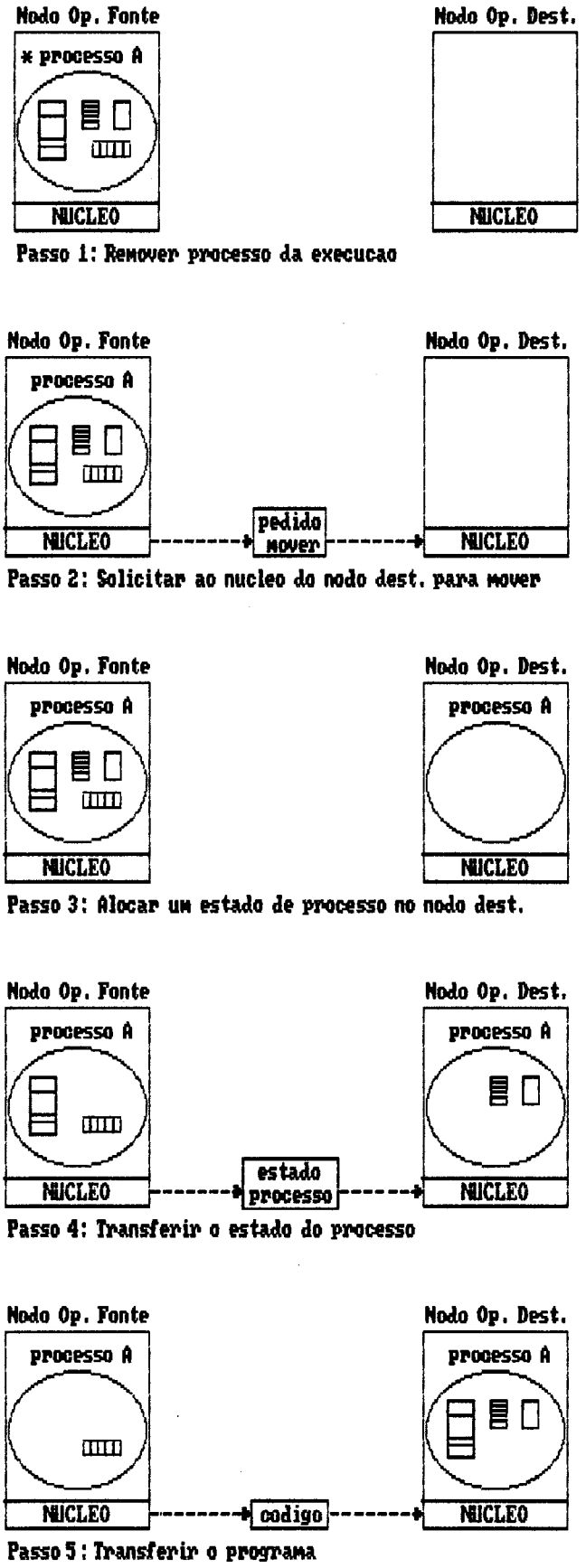
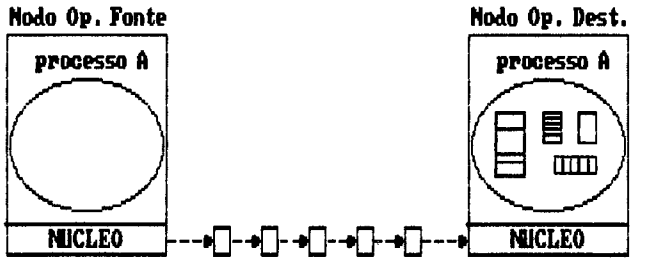


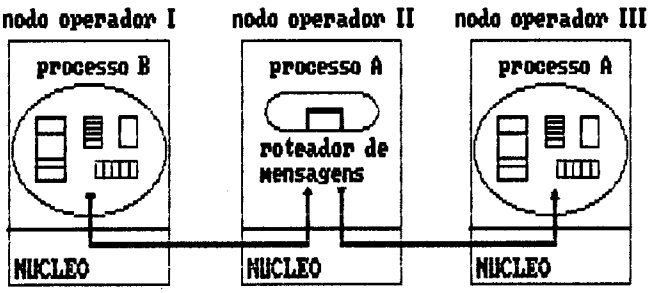

Figura IV.7: Passos para migrar um processo ( POWEL e MILLER ( 1983 ) - continua na página seguinte




Passo 6: Enviar as mensagens pendentes

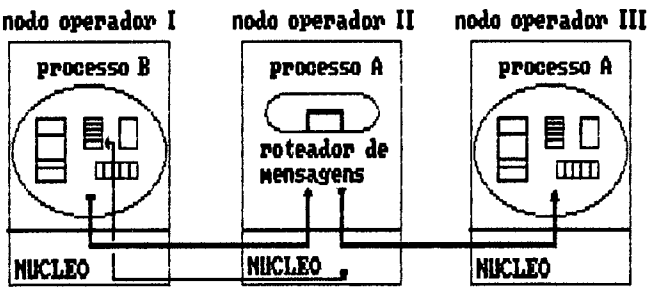

Passo 7: Limpar o estado do processo


Passo 8: Reiniciar o processo




Funções do roteador de mensagens:

- redirecionar a mensagem para a nova localização do processo;




- e enviar mensagem para o núcleo do processo emissor o novo endereço, para que este atualize as suas tabelas de endereços dos processos.

Figura IV.7: Passos para migrar um processo ( POWELL e MILLER ( 1983 )

Nesse mecanismo para migração de processos, o problema é quando se pode eliminar o roteador de mensagens, criado no passo 7, que substitui o processo migrado. Os autores optaram por deixar para sempre o roteador de mensagens, considerando que essa forma é mais segura, do que estimar um tempo para sua eliminação, além do que o espaço de memória exigido para retê-lo é bastante pequeno; o que é perceptível.

## IV.5 ÓRFÃOS

Muitas das técnicas aqui discutidas para a estruturação de Sistemas Distribuídos tolerante a faltas, deixam para um segundo plano; ou melhor, admitem a existência de um suporte altamente confiável para a sincronização e comunicação entre processos.

Pela vasta literatura existente, sabe-se que o tipo de mecanismo para ser utilizado como suporte para tal necessidade, é aquele baseado em troca de mensagens; pois este não exige o compartilhamento de memória como o meio para o estabelecimento de uma comunicação ( veja, por exemplo, SEGRE e KIRNER ( 1982 ) ou a breve discussão a respeito desse assunto descrito no capítulo V ).

Percebe-se, entretanto que, independentemente da forma de implementação de mecanismos de sincronização e comunicação entre processos baseados no conceito de troca de mensagens, eles se apóiam em outros mecanismos de mais baixo nível para acessarem o meio físico de comunicação ( protocolos de comunicação ), por onde realmente as informações fluem de um nodo operador para um outro, para o estabelecimento da comunicação entre os processos residentes nesses nodos. Tal meio físico, como já mencionado, é uma rede local, que para o momento a topologia é irrelevante. Como também é, o tipo de protocolo desse nível que poderia ser utilizado. Pois, de um modo geral, um protocolo não se preocupa em tolerar faltas do tipo colapso do nodo operador destino ou pior, colapso no nodo operador emissor de mensagem, ou etc., uma vez que seus objetivos são: primeiro, serem independentes do tipo da aplicação; segundo, estabelecer regras de como acessar o meio físico de comunicação; e terceiro, tolerar corrupção de mensagens em tramitação. Para maiores detalhes, veja referências a respeito de protocolos de comunicação de redes de computadores.

Diante disso, como hipótese inicial, pode-se admitir os seguintes tipos de problemas que podem surgir durante uma comunicação entre dois nodos operadores fins ( nodo emissor e nodo receptor destino ):

- atrasos na tramitação das mensagens, devido a congestionamento na rede de comunicação ou devido a corrupção e conseqüente retransmissão ( tolerância a esse tipo de falha );
- perda da mensagem por falha transiente de algum nodo de comunicação da rede;
- colapso do nodo operador receptor da mensagem;
- colapso do nodo operador emissor da mensagem;
- colapso de nodos de comunicação ou rompimento da linha de comunicação. Cabe observar que esse tipo de problema é dependente da topologia da rede de comunicação. Caso o problema implique no isolamento do nodo operador, então, a falta é do mesmo tipo que os dois anteriores. Caso ele leve ao colapso da própria rede, então, nada se pode fazer.

Dessa forma, é muito interessante que os mecanismos baseados em troca de mensagens, desconfiem da possibilidade de ocorrência de algum problema desse tipo e comunique ao processo que os está utilizando, a fim de que este tome alguma providência a respeito.

Obviamente, que a detecção de eventos indesejáveis é importante, pelo menos, para troca de mensagens síncrona e semi-síncrona, se se desejar que um processo não fique eternamente esperando por alguma mensagem que nunca chegará, pois pode ter ocorrido que o nodo operador alvo esteja com defeito ( entrou em colapso ). A forma até hoje utilizada para se desconfiar é por meio do estabelecimento de um tempo limite para que seja efetuada a comunicação. Quando um processo envia de forma síncrona, uma mensagem para um outro processo remoto, se define um tempo limite para que ela chegue ao seu destino. Caso não chegue nenhuma confirmação desse objetivo dentro do tempo fixado, apenas dá para se desconfiar que pode ter ocorrido algum problema dos tipos mencionados. Note, porém, que se o tempo limite não for bem estabelecido, pode-se incorrer em um erro não existente. Por exemplo, a mensagem pode ter chegado ao seu destino e este enviado uma confirmação, contudo, esta se atrasou; por motivo de congestionamento ou corrupção da mesma e conseqüente retransmissão pelo próprio mecanismo do protocolo utilizado.

O método já consagrado para corrigir falhas do tipo transiente ( principalmente, atraso e perda ) é a retransmissão da mensagem, que

é feita quando o tempo limite estabelecido para a recepção da resposta estoure. Caso as retransmissões não tenham sucesso ( naturalmente, fixa-se, também, um limite de número de retransmissões ), então a falha pode ser proveniente de alguma falta do tipo colapso do nodo operador destino ou devido a alguma falha não prevista no processo responsável pela emissão da resposta, ou etc.

Diante do fato de que a primeira medida a ser tomada pelo mecanismo de comunicação entre processos, na ocorrência de não chegar uma resposta à mensagem enviada dentro do prazo máximo estabelecido, é a sua retransmissão, é preciso saber se isso não ocasionará problemas.

Preocupado em estudar se o método pode ser aplicado sem nenhuma preocupação ou se deve tomar alguns cuidados, NELSON ( 1981 ) pesquisou e definiu as semânticas de chamadas remotas de procedimentos ( um mecanismo de alto nível de abstração para a comunicação entre processos ), e os problemas que elas podem ocasionar. Diante do não esgotamento desse assunto, principalmente ao se levar em consideração fatores como desempenho, eficiência, tipo de subsistema de comunicação utilizado, tipo de aplicação, etc., vários outros pesquisadores deram prosseguimento, como SPECTOR ( 1982 ), SHRIVASTAVA e PANZIERI ( 1981, 1982 ), SHRIVASTAVA ( 1983 ), PANZIERI e SHRIVASTAVA ( 1982, 1985, 1988 ), BIRREL ( 1985 ), etc.

Apenas para ilustrar, agora, uma chamada de um procedimento remoto, pode implicar em mais de uma chamada efetiva, desde que não chegue uma resposta dentro de um intervalo de tempo pré-definido pelo procedimento chamado. Assim, um procedimento chamado pode ser executado mais do que uma vez, mesmo que tenha sido feita uma única solicitação pelo programa usuário. Se o referido procedimento faz operações incrementais, certamente, a sua reexecução levará o sistema a cometer um erro grave. Como descrito no item III.4.3, é sempre melhor evitar falhas previstas ou corrigir os erros logo que são detectados, a fim de eles não interfiram nas computações corretas que seguem.

Além desse problema, existem outros decorrentes dos vários tipos de problemas acima citados, que podem dar o surgimento de operações tão indesejáveis quanto aquelas. É comum na literatura sobre tolerância a falhas, denominar essas operações de órfãos, que segundo SHRIVASTAVA ( 1983 ), se deve a B.LAMPSON.

Como uma definição, órfãos são computações de um Sistema Distribuído que são escaladas devido a ocorrência de vários eventos

indesejáveis tais como duplicação de mensagens e colapsos de nodos operadores ou de comunicação. E que, tais computações devem ser detectadas e removidas do sistema, pois no mínimo, elas desperdiçam recursos, podendo até interferir nas tarefas de processos corretos ( SHRIVASTAVA, 1983 ).

A seguir descreve-se, de forma resumida, tanto os modelos de faltas como as técnicas sugeridas e algumas implementadas, encontrados na literatura. Mais precisamente, descrever-se-á a devida a PANZIERI e SHRIVASTAVA ( 1985, 1988 ), cuja pesquisa iniciou-se em 1982 ( PANZIERI e SHRIVASTAVA, 1982 ), SHRIVASTAVA ( 1983 ) e que se preocupou com a implementação, levando em consideração as implementações já existentes, como em Cedar ( BIRREL e NELSON ), Sun ( Sun Microsystem ) e Courier-Rpc ( Xerox System Integration Standard ), mencionadas nessa referência.

O mecanismo de chamada remota de procedimento ( RPC - Remote Procedure Call ), foi denominado de Rajdoot, projetado para Sistemas Distribuídos consistindo de "clientes" e "servidores" e que provê um conjunto conveniente de primitivas que podem ser utilizadas por esses clientes e servidores, arbitrariamente.

Com vistas de que tal mecanismo possa ser aplicada a nível comercial, os pesquisadores se preocuparam também com as questões de desempenho.

De suas análises, PANZIERI e SHRIVASTAVA ( 1985, 1988 ) chegaram à conclusão de que, para o projeto de uma RPC são necessárias tomar as seguintes decisões principais:

1) seleção da semântica;
2) seleção das capacidades de tolerância a faltas - sob que condições uma chamada pode terminar de forma normal;
3) provisão de facilidades para o tratamento de órfãos como requeridas pela semântica escolhida;
4) funcionalidade do serviço de transporte subjacente e o modelo de execução da RPC.

e resolveram implementar um mecanismo que tem as seguintes características:

a) suporta semântica "exatamente uma vez";
b) permite o aninhamento de chamadas;
c) permite o estabelecimento de tempo limite para chamadas de clientes e repetição de chamadas, e
d) incorpora medidas para tolerar falhas no subsistema de comunicação;

e) incorpora medidas para detectar e eliminar órfãos, gerados devido ao colapso de cliente.

Quanto a colapso de servidor, na sua ocorrência a chamada deve terminar de forma anormal, ficando para o cliente ou o servidor, quando restabelecido, tomar as devidas medidas para restaurar o sistema, recolocando-o em um estado seguro.

A seguir descreve-se de forma resumida as decisões tomadas para a definição das características de tal mecanismo de RPC.

O modelo físico de Sistema Distribuído adotado foi o de ser uma coleção de nodos de processamento conectados por um subsistema de comunicação e as faltas em tal subsistema são os responsáveis pelos seguintes tipos de falhas:

1) uma mensagem transmitida de um nodo não chega ao seu destino;
2) as mensagens não são recebidas na mesma ordem em que elas foram enviadas;
3) uma mensagem se corrompe durante a sua transmissão; e
4) uma mensagem pode ser replicada durante a sua transmissão.

Desses tipos os pesquisadores consideraram as duas primeiras e a quarta, coletivamente, como falha de comunicação, a qual deve ser tratada, e a segunda considerou que as mensagens carregam informações suficientes que permitem, com segurança, que o receptor descarte as corrompidas ( por exemplo, "checksum" - técnica bastante conhecida ).

O modelo de falta para falhas de nodos de processamento foi o seguinte: ou o nodo trabalha de acordo com as especificações ou aquele nodo para de trabalhar. Após um colapso, um nodo é reparado dentro de um tempo finito e colocado novamente em operação.

PANZIERI e SHRIVASTAVA ( 1985, 1988 ) admitiram as seguintes condições sob as quais são possíveis um término normal de suas RPC.

a) durante uma chamada não ocorrem falhas de nodos e/ou de comunicação;
b) o mecanismo do RPC compete com um número fixo de falhas de comunicação;
c) o mecanismo do RPC compete com um número fixo de falhas de comunicação e de colapso de nodo servidor;
d) o mecanismo do RPC compete com um número fixo de falhas de comunicação, de nodo servidor e de nodo cliente.

Diante dos protocolos de rede tipicamente empregarem limites de tempos para evitar que um processo que está a espera de uma

mensagem fique neste estado indefinidamente, assumiu-se que um processo cliente a espera de resultados do servidor chamado, tenha um limite de tempo, ou equivalentemente, algum mecanismo dependente de protocolo, que sinalize o cliente se nenhuma resposta seja recebida após algum tempo. Se uma chamada termina anormalmente ( expira o limite de tempo ), então existem quatro possibilidades mutualmente exclusivas a considerar:

a) o servidor não recebeu a mensagem enviada;
b) a mensagem de resposta não chegou ao cliente;
c) o servidor entrou em colapso durante a execução da chamada e ou permaneceu assim ou não reassumiu a execução após a recuperação do colapso; e
d) o servidor está ainda executando o trabalho solicitado, cujo caso, a execução poderia interferir nas atividades subseqüentes do cliente, como ilustra a figura IV.8.

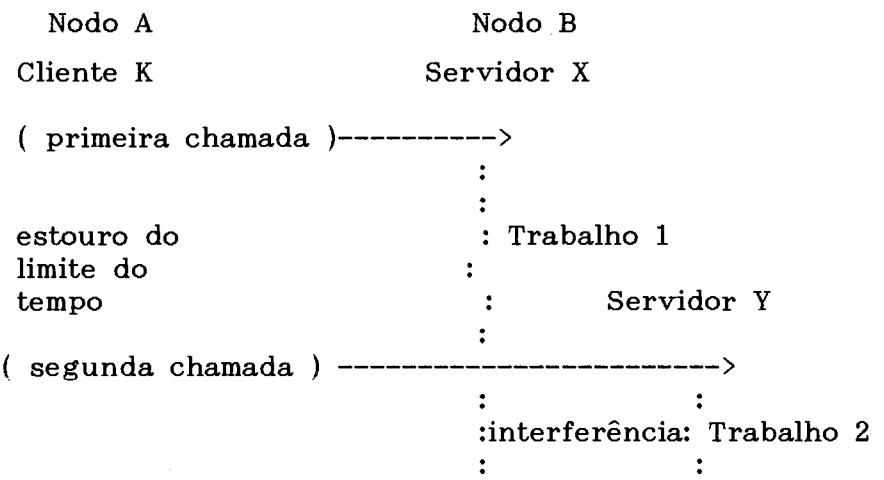



Figura IV.8: Exemplo de Interferência provocada pelo estouro do limite de tempo

O cliente K residente no nodo A emitiu uma chamada para o servidor X residente no nodo B, para que executasse o trabalho 1. Porém, a chamada terminou anormalmente antes que X completasse o trabalho. O cliente então emitiu uma outra chamada para algum servidor Y também residente no nodo B. Se as computações de X estiverem em andamento e os trabalhos 1 e 2 têm dados em comum, então essas computações podem interferirem entre si - o trabalho 1 é órfão, como não foi eliminado antes de se iniciar o trabalho 2, ele interferiu na execução deste. Note que esse tipo de concorrência é indesejável, desde que a execução de

um programa seqüencial deverá dar surgimento a uma computação seqüencial, caracterizada por um único fluxo de controle. As técnicas de controle de concorrência ( por exemplo, região crítica ), foram elaboradas para evitar interferências entre programas distintos sob a hipótese de que cada um executará uma computação seqüencial.

Tal tipo de interferência, pode ocorrer nos casos de colapso do cliente A ou do servidor B ( veja a referência seguida para os detalhes e outros exemplos ilustrativos )

Como os órfãos podem ser tratados? Isso dependerá da semântica utilizada.

Quanto às semânticas largamente aceitas, a saber NELSON ( 1981 ), SPECTOR ( 1982 ), etc.:

- "pelo menos uma vez" ( at least once ): uma chamada terminada normalmente implica que o servidor executou o trabalho solicitado uma ou mais vezes;
- "quanto muito uma vez" ( at most once ): esta tem semântica idêntica à "exatamente uma vez" se a chamada tiver sucesso, caso contrário, ela implica que não deverá produzir quaisquer efeitos colaterais.
- "exatamente uma vez" ( exatly once ): esta semântica implica que uma chamada terminada normalmente, resultou na execução do trabalho solicitado uma única vez; e

Independentemente da semântica a ser utilizada, SHRIVASTAVA ( 1983 ) em sua pesquisa anterior, definiu as duas regras que um servidor deve obedecer para que possam ser vistas como ocorrências atômicas a despeito da presença de órfãos, que são:

Regra 1: uma vez iniciada a execução de uma chamada, as computações pertencentes às chamadas anteriores ( se existirem ) nunca deverão ser permitidas de serem escaladas ( isto é, todos os órfãos das chamadas anteriores devem preceder desta chamada ).

Regra 2: computações locais realizadas pelos processos trabalhadores de um servidor como resultado de uma única chamada, não devem interferir uma com as outras.

Diante disso, PANZIERI e SHRIVASTAVA ( 1985, 1988 ) descartaram daquelas semânticas citadas, as duas primeiras, pelos seguintes motivos:

- a semântica "pelo menos uma vez", por permitir que um serviço solicitado por uma única chamada seja feita uma ou mais vezes, assim, ela pode desobedecer as regras 1 e 2, salvo se o serviço é idempotente;

- a semântica "quanto muito uma vez", por exigir que o serviço seja executado uma única vez ou não faça nada e não deixe resíduos, há a necessidade de se detectar e eliminar órfãos, para obedecer as regras 1 e 2, e também é necessário um sofisticado suporte de recuperação de erros para trás, para desfazer dos efeitos colaterais da chamada ( para não deixar resíduos ). Uma RPC que obedece essa semântica é descrita por LIN e GANNON ( 1985 ), a qual foi denominada de Chamada Remota Atômica de Procedimento.

Tendo em vista o objetivo de se construir um mecanismo de RPC que possa ser utilizado para aplicações em geral, independentemente se elas usam ou não ações atômicas ( que é justamente a exigência da semântica "quanto muito uma vez" ).

Diante disso, o mecanismo proposto optou pela adoção da semântica "exatamente uma vez" e para que ela obedeça a regra 2, deve-se garantir que órfãos serão detectados e eliminados antes que uma nova chamada seja feita. Assim, admite-se que o mecanismo fica suficientemente neutro, podendo suportar aplicações que usam ou não ações atômicas e que os serviços chamados podem ou não serem idempotentes.

PANZIERI e SHRIVASTAVA ( 1985, 1988 ) preocupados também com o fator desempenho, se detiveram em averiguar os tipos de protocolos de rede que melhor características ofereciam para os seus propósitos. Acabaram optando pela adoção do protocolo a nível de transporte tipo datagrama de tamanho praticamente ilimitado.

Quanto à capacidade de tolerância a faltas que o mecanismo de RPC deveria conter para que uma chamada possa ser considerada terminada de forma normal, das quatro condições acima citadas, foi incorporado somente a segunda ou seja, ele compete com um número fixo de falhas de comunicação. Isso pelo seguinte:

- quanto à quarta ( condição d ), ou melhor quanto a colapso de cliente e de servidor, foi descartada de imediato sobre a base de que ela provê muita funcionalidade e é muito complexa para se implementar. Optou-se, por considerar melhor, em deixar para o processo cliente, implementar suas próprias estratégias de resistência a colapso, em vez de fixá-las a nível de RPC;
- quanto a terceira ( condição c ), ou melhor quando da ocorrência de colapso de servidor, preferiu-se optar em fazer com que a chamada terminasse anormalmente. Isso porque não há garantia de que as operações efetuadas antes do colapso tenham sidos destruídas por

ele, a menos que sejam empregados procedimentos de recuperação apropriados. Assim, se um servidor após o seu colapso reiniciar a sua execução, o efeito final poderia não ser ao devido a uma única execução e sim a mais que uma. Diante disso, escolheu-se a solução mais simples que é evitar a ocorrência de tais situações, insistindo que a chamada termine anormalmente.

Além do mecanismo conter medidas para tolerar faltas do subsistema de comunicação, terá capacidade para tratar órfãos devido a colapso de clientes.

Para tanto, os servidores terão capacidade para detectar que o cliente tinha entrado em colapso e depois se recuperou ou não e tomar as devidas medidas que serão:

- se o cliente após o colapso se recuperar e fizer uma chamada para executar o mesmo serviço, pois a anterior não foi considerada terminada, o servidor chamado deverá antes eliminar todos os órfãos oriundos da chamada anterior, para depois executar esta;
- se o cliente após o colapso não fizer mais nenhuma chamada para executar o mesmo serviço, é garantido que todos os órfãos serão detectados e eliminados.

A primitiva disponível para os clientes para solicitar uma chamada remota, onde os parâmetros e resultados são passados por valor, é a seguinte:

```
rpc( servidor: ...; call: ...; timeout: ...; retry: ...;
     var reply: ...; var:= rpc_status: ... );
```

onde o primeiro parâmetro especifica o servidor da operação; o segundo especifica o nome da operação e seus parâmetros relevantes; o terceiro parâmetro especifica o tempo limite para cada chamada na tentativa de conseguir a execução da operação pedida; o quarto indica o número de vezes que a chamada pode ser tentada para atingir o objetivo do parâmetro anterior ( o valor de omissão é zero ); o parâmetro reply, é para receber a resposta da operação pedida; e por último, o parâmetro rpc_status é também um de resposta, que pode assumir um dos seguintes valores: rpc_status = ( OK, NOTDONE, UNABLE ).

O valor da variável rpc_status, que como o seu identificador sugere, oferece após o término da chamada, informações sobre a execução da chamada remota e, conseqüentemente, sobre a variável

reply; se ela realmente contém uma resposta ou o seu conteúdo não tem nenhum significado. Cada valor de rpc_status expressa o seguinte:

- rpc_status = OK: o serviço chamado foi executado exatamente uma única vez pelo servidor e o resultado está disponível em reply. Isso representa uma terminação normal de uma chamada que tomou no máximo ( n+1 ) * t unidades de tempo, onde n é o valor do número máximo de tentativas ( o quarto parâmetro ) e t o tempo limite para cada tentativa ( o terceiro parâmetro );
- rpc_status = NOTDONE: a chamada não foi executada. Essa resposta é obtida quando alguma falha de comunicação não permitiu que a mensagem de chamada fosse transmitida ao servidor; tal resposta é obtida em menos do que t unidades de tempo.
- rpc_status = UNABLE: o servidor chamado pode ter no máximo executado a operação chamada uma única vez. Entretanto, o parâmetro reply não contém nenhuma resposta. Esse caso representa um término anormal de chamada que gastou no máximo ( n+1 ) * t unidades de tempo. É garantido que qualquer computação que a chamada possa ter gerado, também terminou. Assim, quando a primeira chamada do cliente termina, não existirão computações em andamento, tanto no nodo que contém o servidor chamado como também nos servidores que aquele possa ter chamado como conseqüência ( chamada aninhada ).

Os mecanismos suporte dessa primitiva, em síntese, são:

M1: cada mensagem é acompanhada de um número gerado de forma seqüencial por meio de um relógio a prova de colapso contido em cada nodo de processamento, por meio do qual é possível evitar a execução de chamadas repetidas ou chamadas do passado; além disso, é utilizado como protocolo de comunicação a nível de transporte o datagrama com comprimento de mensagem variável - isso evita a necessidade da concatenação ordenada das partes constituintes de uma mensagem - acompanha também cada mensagem um "checksum" para permitir que o receptor descarte mensagens corrompidas;

M2: cada chamada contém um "deadline" indicando ao servidor o tempo máximo disponível que ele tem para executar a operação pedida. O deadline é igual a ( n+1 ) * t unidades de tempo. Note que esse tempo é igual ao tempo máximo que o chamador espera por uma resposta. Se o "deadline" se expira, então o servidor aborta a execução e a chamada termina anormalmente. Percebe-se assim

que, se não ocorrer colapso de nenhum nodo do sistema, M2 será o suficiente para competir com órfãos. Os dois mecanismos restantes competem com órfãos.

M3: cada nodo mantém um contador estável ( a prova de colapsos ), denominado de "crash-counter" ( contador de colapsos ), que é incrementado imediatamente após a recuperação de um nodo de seu colapso. Um nodo também mantém uma tabela de valores de "crash-counter" dos clientes que fizeram chamadas para ele. Uma mensagem de chamada contém o valor do "crash-counter" do cliente. Se esse valor for maior do que o correspondente na tabela do servidor, então poderiam existir órfãos os quais são primeiro abortados antes de dar prosseguimento à chamada.

M4: cada nodo tem um processo exterminador que ocasionalmente verifica os valores dos "crash-counters" dos outros nodos ( por meio da transmissão de mensagens e recepção das respostas ), e aborta quaisquer órfãos quando ele detectar quaisquer colapsos.

Note que se a rpc_status retornar com o valor UNABLE, o que significa que a chamada terminou de forma anormal, é possível que no nodo servidor ficou algum resíduo da chamada ( operações órfãs ), pois o que os mecanismos garantem é que não existirão computações em andamento devido à chamada. Assim, caso a operação pedida não seja idempotente, é necessário que o cliente ou o próprio servidor chamado, tome as devidas providências para evitar que esses resíduos não influam no progresso do processamento. Para maiores detalhes veja PANZIERI e SHRIVASTAVA ( 1985, 1988 ) e HERLIHY e McKENDRY ( 1989 ), que propõem duas versões de um método para a detecção e eliminação de órfãos baseado em relógio. A primeira em um relógio de tempo real e a outra em um relógio lógico, sendo que esta se assemelha em muito com a técnica da utilização do "deadline". Uma referência preocupada com a coleta de lixo necessária para desocupar espaço de memória dos órfãos eliminados, é a MANCINI e SHRIVASTAVA ( 1988 ).

# CAPÍTULO V

## SISTEMAS OPERACIONAIS DISTRIBUíDOS

Como descrito nos capítulos anteriores, o "software" de um Sistema Distribuído continua sendo um dos grandes obstáculos para o seu desenvolvimento a nível de aplicações em grande escala comercial. Principalmente, de Sistemas Distribuídos para propósitos gerais, mais comumente denominado de Sistema Operacional Distribuído. Isso se deve ao seu porte e complexidade, diante das necessidades de atuarem em ambientes de múltiplos computadores ou múltiplos processadores, de procurarem dar aos seus usuários a visão de estarem trabalhando em uma única máquina, de oferecerem a confiabilidade esperada pelos seus usuários para o processamento de seus programas e que tenham uma disponibilidade maior que a encontrada em Sistemas Operacionais não distribuídos.

Toda essa complexidade foi demonstrada nos capítulos anteriores. Assim, o presente objetiva descrever apenas o estudo sobre Sistemas Operacionais Distribuídos no que refere-se ao seu "software" mais básico, ou seja, quanto aos tipos de núcleos, suas funções e seus mecanismos para comunicação e sincronização entre processos.

Adicionou-se o levantamento bibliográfico a respeito dos tipos de Sistemas Operacionais Distribuídos existentes na literatura e uma breve análise sobre as linguagens de alto nível propostas ( algumas implementadas ) para a programação de sistemas desse tipo, a fim de complementar os capítulos anteriores a respeito dos tópicos de interesse direto a este trabalho.

## V.1 NÚCLEO

Segundo o item III.2, uma forma de estruturação de Sistemas Operacionais é por meio de camadas hierárquicas, cada uma oferecendo recursos computacionais de mais alto nível, apoiada sobre os recursos da camada inferior. Um outra forma seria a estruturação por meio de extensões das camadas inferiores, mais precisamente, da camada mais próxima do "hardware" subjacente; a camada de mais baixo nível.

Uma grande vantagem de estruturar um Sistema Operacional em uma dessas formas ( preferencialmente em camadas hierárquicas ), é a aplicação do desenvolvimento de projeto segundo os métodos de cima para baixo ( "top down" ) ou de baixo para cima ( "bottom up" ), bastante conhecidos e difundidos nos meios científicos relacionados à Ciência da Computação.

Independentemente se o sistema será estruturado em camadas hierárquicas ou estendidas, existe a primeira, aquela que ficará em contacto com os recursos de "hardware" existentes. Essa camada é normalmente denominada de núcleo básico do sistema, ou seja a primeira máquina virtual, que tem como objetivo esconder os detalhes da máquina subjacente e oferecer recursos computacionais ( primitivas - operações - e estruturas de dados suporte ) mais adequados aos propósitos das camadas superiores ( de maior abstração ). Especificou-se como primeira máquina virtual, uma vez que um sistema estruturado em camadas hierárquicas, cada uma define uma máquina virtual para a camada superior ( máquina virtual de nível mais alto ).

Como a preocupação neste item é a descrição das primitivas de um núcleo para Sistemas Operacionais Distribuídos sem a definição exata quanto aos seus fins, não se dará atenção aos aspectos como tolerância a faltas de "hardware" ou de "software", delineados nos capítulos anteriores, como também, às topologias de redes locais e os protocolos de comunicação; estes por fugirem dos objetivos do trabalho e aqueles

por serem especificidades dos que serão brevemente discutidos a seguir.

### V.1.1 NÚCLEO DESCENTRALIZADO E NÚCLEO REPLICADO

De acordo com o capítulo II, um Sistema Distribuído caracteriza-se pelo seu controle estar descentralizado, residentes nos vários nodos operadores interligados por um subsistema de comunicação. Adotando-se a forma de estruturação de um Sistema Operacional Distribuído em camadas hierárquicas, o seu núcleo pode ser construído de forma distribuída ou replicada. Na forma distribuída, em cada nodo operador constroi-se um núcleo que ofereça as primitivas necessárias para a implementação das funções das suas camadas superiores particulares e, na outra forma, constroi-se um núcleo como se o sistema fosse centralizado e replica-o para cada nodo operador.

No caso em que todos os nodos operadores contenham núcleos que são réplicas de um único, poderá acontecer de que, para um ou mais nodos operadores, seus núcleos ofereçam muito mais recursos que os necessários para o suporte das funções de suas camadas superiores, dentro da distribuição definida para o Sistema Operacional Distribuído. Conseqüentemente, isso implicará em um desperdício de memória em cada um desses nodos. Entretanto, traz a vantagem da homogeneidade. Como cada nodo operador oferece uma máquina virtual idêntica às dos outros nodos, pode-se atribuir a ele quaisquer funções independentes de características específicas de "hardware". Exceção deve ser feita quando a função necessita de algum dispositivo específico que sequer possa ser simulado. Por exemplo, há a necessidade de uma impressora para se poder implementar um servidor de impressão.

No caso de núcleo distribuído, cada nodo operador do Sistema Distribuído deverá possuir um específico, para dar o suporte apenas às suas funções como parte das que deverão serem exercidas pelo Sistema Operacional Distribuído. Obviamente, algumas funções deverão ser replicadas em cada núcleo de cada nodo operador. Umas bem visíveis são aquelas exercidas pelo mecanismo de comunicação, no mínimo, entre os núcleos. É perceptível que essa forma de construção do núcleo para um Sistema Operacional Distribuído, visa que ele siga o modelo de nodos clientes x nodos servidores.

Nesse modelo clientes x servidores, de um modo geral, define-se dentre o conjunto de nodos operadores do Sistema Distribuído, aqueles que terão as tarefas específicas de prestarem os serviços de impressão e de arquivo ( nodos servidores ) para o Sistema Operacional, e aqueles que serão destinados à execução de processos de clientes ( nodos clientes ). Se assim definido, note que, os nodos servidores podem ser de porte bem menor do que os outros, pois, de um modo geral, esses tipos de periféricos apresentam uma velocidade de acesso muito menor do que a de processamento daqueles. Além disso, nesses nodos suas capacidades de memórias podem ser bem definidas, considerando que eles deverão executar apenas os serviços determinados pelo Sistema Operacional. Dessa maneira, nesse modelo, é comum construir Sistemas Distribuídos composto . pelo menos com dois tipos de computadores: um para servir como nodos clientes e o outro para servir como nodos servidores. A desvantagem desse modelo em relação ao outro é quanto à flexibilidade, ou seja, nesse ficam estaticamente estabelecidos os nodos operadores que exercerão as funções de servidores de impressão e de arquivo.

Em termos de tolerância a falhas em Sistemas Operacionais Distribuídos, entretanto, é visível que outros fatores podem ser mais determinantes, que a opção por um ou outro modelo de construção de núcleo se torna secundária. Por exemplo, pode se citar: colocação de redundância de servidores de arquivo e/ou de impressão, mecanismos para migração de processos, etc., como vistos no capítulo IV.

Percebe-se, entretanto que, em termos de custo, a vantagem de uma forma de construção de núcleo para Sistema Operacional Distribuído pode representar uma desvantagem em relação à outra. Poderia-se, então, encontrar um meio termo. No modelo clientes x nodos servidores, especificar alguns nodos clientes como potenciais para exercerem as funções daqueles e, conseqüentemente, deixar o supervisor do sistema ciente disso. Isso poderia ser feito de duas maneiras:

(a) cada nodo potencial contém as funções completas; ou

(b) cada nodo contém apenas o núcleo em potencial, necessitando para se tornar completo, que os processos componentes da função sejam migrados para ele.

Observa-se que, na forma de núcleo replicado, isso pode também ser feito, com a vantagem de poder ser para qualquer nodo ( com potencial de "hardware" necessário ), bastando que se migre para ele os processos componentes da função desejada.

Diante disso, pode-se concluir que, de uma maneira geral, é mais interessante construir um núcleo replicado para Sistemas Operacionais Distribuídos.

## V.1.2 MECANISMOS DE COMUNICAÇÃO E SINCRONIZAÇÃO SEM COMPARTILHAMENTO DE MEMÓRIA

Todo Sistema Operacional necessita de algum mecanismo por meio do qual seus processos possam cooperarem e competirem pela utilização otimizada dos recursos computacionais de forma harmônica, para alcançar seus objetivos com o melhor desempenho possível.

Como é de conhecimento, os primeiros mecanismos elaborados e implementados para esses fins, foram para sistemas que possuíam uma única memória compartilhada. Mecanismos esses já bastante difundidos, desde aqueles que oferecem primitivas simples como "wait" e "signal", devida a DIJKSTRA ( 1968 ), até os mais estruturados como o monitor, devida a HOARE ( 1974 ). Entretanto, como os objetivos desse item são com relação aos mecanismos de comunicação e sincronização entre processos de um Sistema Distribuído, onde a cooperação necessita ser feita entre processos residentes em nodos distintos, aqueles se tornam ineficientes ou até inviáveis, pois para esses casos não há a obrigatoriedade de que os processos compartilhem a memória para poderem cooperar. Agora, eles terão que trocar informações por meio de envio e recepção de mensagens, através do subsistema de comunicação ( uma rede local ) que interliga todos os nodos operadores, para cooperarem. Portanto, não se descreverá nada a respeito daqueles mecanismos, aos interessados recomenda-se as referências mais recentes como as SEGRE ( 1981 ), SEGRE e MENDES ( 1981 ), KIRNER ( 1986 ), KIRNER e MENDES ( 1988 ), LORIN e DEITEL ( 1981 ), PETERSON ( 1985 ), DEITEL ( 1984 ), SILBERSCHATZ e PETERSON ( 1989 ), etc, enfim os livros e artigos sobre conceitos de Sistemas Operacionais.

Naturalmente, mesmo em Sistemas Distribuídos, existirão processos que deverão cooperar e/ou competir que residem em um mesmo nodo operador. Para esses caso, poderia-se utilizar aqueles mecanismos. Porém, em favor da simplicidade, geralmente um núcleo de Sistema Operacional deve oferecer um determinado tipo de mecanismo de comunicação e sincronização. Assim, é comum que as primitivas oferecidas por um núcleo sirvam tanto para comunicação e sincronização

entre processos residentes em um mesmo nodo operador, como também para aqueles residentes em nodos distintos.

Diante das bases principais, distintas entre esses dois tipos de mecanismos, aqueles primeiros foram rotulados de mecanismos de comunicação e sincronização entre processos por meio de compartilhamento de memória e esses de mecanismos de comunicação e sincronização entre processos por meio de troca de mensagens ( KIRNER e MENDES, 1988 ). Como já mencionado, essa denominação distinta, não implica que as primitivas baseadas em um ou outro mecanismo, efetuem operações cujos efeitos sejam totalmente diferentes, uma vez que eles foram elaborados para resolverem os mesmos tipos de problemas, cooperação e competição entre processos.

A seguir descrevem-se os tipos de mecanismos de comunicação e sincronização baseados no conceito de troca de mensagens. Considerando-se que em SEGRE ( 1987 ) e em KIRNER e MENDES ( 1988 ) estão descritos um estudo detalhado sobre eles, apenas se delineará a respeito a fim de servir como futuras referências nos capítulos que seguem.

### V.1.2.1 MECANISMOS DE COMUNICAÇÃO E SINCRONIZAÇÃO BASEADOS EM TROCA DE MENSAGENS

Em termos gerais, como já observado, qualquer forma de comunicação entre dois processos baseado em troca de mensagens, um tem que enviar a mensagem ( processo emissor ) e o outro tem que recebê-la ( processo receptor ). Para tanto, tem que se definir duas primitivas ( enviar e receber ), segundo uma forma bem determinada, para que eles possam, por meio delas, executarem esse tipo de ação.

Diante da possibilidade de que quando um processo enviar uma mensagem o processo receptor não estar pronto para recebê-la, e vice-versa, é necessário definir nessas primitivas como estes processos deverão se comportar ( regras de sincronização ). Por exemplo, se o processo emissor deverá ficar suspenso até que o processo receptor esteja pronto para receber a mensagem. Também, o processo emissor ( ou receptor ) poderia desejar especificar o seu processo parceiro receptor ( ou emissor ), de forma a deixar textualmente claro com quem ele está interagindo ( formas de endereçamento ). Além disso, poderia-se utilizar "buffers" para não comprometer demais o andamento dos

processos emissor e receptor, como o caso do clássico problema dos produtores versus consumidores. Por fim, diante da possibilidade da ocorrência de falhas no subsistema de comunicação e nos próprios nodos ( veja capítulo IV ), pode ser interessante definir nos próprios mecanismos de comunicação formas para detectá-las e tratá-las.

Disso, percebe-se que se pode elaborar vários tipos de mecanismos e várias formas de executar troca de mensagens, dependendo de como são realizados o sincronismo, o endereçamento, a utilização de "buffers" e se envolvem ou não tratamento de falhas ( eventos decorrentes de faltas ). A sincronização refere-se à restrição no andamento dos processos, o endereçamento aos aspectos de identificação dos processos envolvidos na comunicação, podendo ser: implícito ou explícito; simétrico ou assimétrico e direto ou indireto. Com relação à utilização de "buffers", estes poderão estar na origem, no destino ou em ambos os lugares. Por último, quanto ao tratamento de falhas, os mecanismos podem ou não envolver essa característica, podem apenas detectar falhas e sinalizar a ocorrência, deixando para o processo a forma de como tratar, ou eles mesmos podem detectá-las e tratá-las.

Note-se que, mesmo os mecanismos mais simples para esses fins, é de complexidade comparável aos mais estruturados citados no início deste capítulo ( monitores, por exemplo ), pois eles também deverão, no mínimo, estabelecerem uma regra de sincronização e uma forma de comunicação, independentemente da localização dos processos envolvidos; se eles residem ou não em um mesmo nodo - se são locais ou remotos. Isso porque se está interessado naqueles tipos de mecanismos cujas primitivas deverão estar disponíveis a processos de aplicações distribuídas ( Sistema Operacional Distribuído, por exemplo ).

### V.1.2.2 TIPOS DE IMPLEMENTAÇÃO DOS MECANISMOS DE COMUNICAÇÃO E SINCRONIZAÇÃO

#### A) MECANISMOS SÍNCRONOS E ASSÍNCRONOS

Um mecanismo de comunicação e sincronização, como descrito, é implementado por meio de duas primitivas: uma para enviar mensagem e a outra para receber. Segundo a característica de sincronismo, elas podem ser: bloqueante e não bloqueante.

A primitivas enviar bloqueante, ao ser executada por um processo emissor, deverá fazer com que este fique bloqueado até que o processo receptor, remoto ou local, receba a mensagem enviada; só então é que o processo emissor poderá ter continuidade.

A primitiva enviar não bloqueante, apenas encaminhará a mensagem ao núcleo local, para que o processo emissor prossiga com o seu processamento. O núcleo deverá fazer com que a mensagem seja encaminhada ao processo receptor.

A primitiva receber bloqueante, quando executada por um processo receptor, fará com que este fique bloqueado até o recebimento de uma mensagem, após o qual poderá ter continuidade.

A primitiva receber não bloqueante receberá alguma mensagem caso exista uma pronta para ser recebida, caso contrário ela simplesmente terá insucesso na recepção de mensagem. Independentemente do sucesso ou insucesso na recepção da mensagem, a primitiva não bloqueará o processo receptor que a executar.

A implementação dessas primitivas, aos pares correspondentes, exigem algum protocolo que permitam a sincronização e a transferência das mensagens e dependem da estruturação e alocação de "buffers", como também de outros elementos auxiliares no sistema. Isso melhor se notará através do exemplo dado a seguir.

1) **Enviar:**
   - colocar a mensagem a ser enviada em um "buffer" com capacidade para apenas uma;
   - transferir a mensagem desse "buffer" para aquele que o processo receptor destino irá verificar se existe ou não mensagem a ser recebida, por meio da execução da primitiva receber. Em seguida suspende o processo emissor ( bloqueia ), até que realmente o processo receptor receba a mensagem e confirme essa ocorrência a ele. Para isso, verificar se o processo receptor está ativo, isto é, não está suspenso a espera de uma mensagem, caso em que ele deverá ser desbloqueado.

2) **Receber:**
   - verificar se existe alguma mensagem no "buffer" com capacidade para uma mensagem;
   - se existir, transferir a mensagem desse "buffer" ao processo receptor e sinalizar o processo emissor ( confirmação da recepção da mensagem );
   - se não existir, simplesmente suspender o processo receptor.

Nesse exemplo, usou-se "buffer" para a implementação de uma comunicação síncrona. Porém, no caso, nota-se que não haveria tal necessidade, uma vez que a transmissão poderia ser efetuada somente quando ocorresse a sincronização; quando ambos os processos estiverem suspensos, o emissor querendo enviar uma mensagem e o receptor querendo receber, a mensagem seria passada transmitindo-a do espaço de endereçamento do emissor para o do receptor - análogo à passagem de parâmetros da chamada de um procedimento, que após feito isso, ambos os processos seriam sinalizados para dar continuidade aos seus processamentos.

## A.1) MECANISMOS SÍNCRONOS

Um mecanismo síncrono é aquele que se utiliza de primitivas de enviar e receber bloqueantes.

Segundo SEGRE e KIRNER ( 1982 ), há três tipos de mecanismos síncronos:

(1) **rendez-vous:** os processos emissor e receptor se sincronizam quando, então, acontece a passagem de mensagem e ambos prosseguem com os seus processamentos;

(2) **rendez-vous estendido:** quando o processo emissor se sincroniza com o processo receptor, ocorre a passagem de mensagem e este ainda pode conter um trecho de programa para fazer qualquer coisa ( principalmente verificar a validade da mensagem e montar uma mensagem de resposta ) que, quando terminado, envia uma resposta e ambos têm seus processamentos continuados; e

(3) **chamada remota de procedimento:** seu comportamento deve ser idêntico à de uma chamada comum de procedimentos locais.

## A.2) MECANISMOS ASSÍNCRONOS

Os mecanismos assíncronos são aqueles que se utilizam de primitivas enviar e receber bloqueantes e não bloqueantes, sendo que pelo menos um deles é sempre não bloqueante.

## B) ENDEREÇAMENTO

O endereçamento refere-se às técnicas usadas para a identificação dos processos envolvidos em uma comunicação. Esse aspecto com vistas às aplicações, é muito importante para a definição do comportamento funcional das primitivas. Por exemplo, para o problema de clientes/servidor ( comunicação vários para um ) é importante que um processo receptor ( no caso o processo servidor ) possa receber mensagens de quaisquer processos que queiram enviar para ele ( processos clientes ). Para esse caso, é interessante que exista uma primitiva receber que não exige a identificação do processo emissor. Já para uma comunicação um para um, talvez seja mais vantajoso usar primitivas enviar e receber que exijam a identificação dos processos receptor e emissor, respectivamente; pois além de deixar explicitado no texto do programa a intenção da comunicação, é muito mais seguro ( evita-se que, por algum erro de programação, ocorra uma comunicação possível, porém, indesejável ). Às vezes é importante a existência no sistema de primitiva de emissão múltipla. Por exemplo, para um sistema de Correio Eletrônico - convocação de uma reunião. E assim por diante. De face à variedade de aplicações que possam existir, descreve-se neste item, as técnicas de endereçamento que se julgam cobrir todas as formas possíveis para a implementação das primitivas enviar e receber.

### B.1) ENDEREÇAMENTO IMPLÍCITO E EXPLÍCITO

Segundo GENTLEMAN ( 1981 ), o endereçamento implícito corresponde ao uso de ligações previamente estabelecidas entre os processos do sistema. Essas ligações, fixadas quando da criação dos processos, são mantidas durante toda a vida do sistema, contribuindo para uma situação estática, raramente aceitável na prática.

O endereçamento explícito baseia-se na utilização de nomes, endereços e outros atributos dos processos, explicitamente referenciados nas primitivas de comunicação. De acordo com ANDREWS e SCHNEIDER ( 1983 ), o endereçamento explícito é classificado como:

- **direto**: quando explicita a identificação do processo, por nome ou endereço, ou etc.

- **indireto:** quando explicita um elemento intermediário ( um porto por exemplo ) o qual pode estar ligado, a princípio, a qualquer ou quaisquer outros processos.

A forma de endereçamento explícito direto é mais simples e mais apropriada para sistemas que terão configuração estática. Enquanto que a outra forma de endereçamento é mais flexível e apropriada quando se deseja que o sistema possa ser reconfigurado dinamicamente.

## B.2) ENDEREÇAMENTO SIMÉTRICO E ASSIMÉTRICO

O endereçamento simétrico caracteriza-se pela exigência das identificações dos processos fonte e destino das mensagens nas primitivas de comunicação.

O endereçamento assimétrico, por outro lado, exige que exista somente a identificação do processo destino da mensagem. No par de primitivas enviar e receber, somente a enviar deverá explicitar a identificação do processo destino, receptor das mensagens enviadas. A primitiva receber, deverá ser do tipo recebe de qualquer um.

Note que, as primitivas com endereçamento simétrico, tornam as comunicações um tanto estáticas sendo, portanto, contra-indicadas em aplicações do tipo cliente/servidor, principalmente se os clientes são variáveis. Enquanto que, as com endereçamento assimétrico, tornam as comunicações mais flexíveis sendo, assim, adequadas para as aplicações daquele tipo, pois permite o não determinismo na ordem da recepção das mensagens e faz com que o servidor possa ser utilizado como rotina de biblioteca, mesmo em reconfiguração dinâmica ( KIRNER e MENDES, 1988 ).

## B.3) COMUNICAÇÃO ENTRE GRUPOS DE PROCESSOS

Segundo KIRNER e MENDES ( 1988 ), o conceito de agrupamento de processos atuando conjuntamente é de vital importância em Sistemas Distribuídos. Esses processos costumam ser utilizados para obtenção de processamento paralelo, aumento de disponibilidade de dados, redução de tempo de resposta, compartilhamento de recursos, aumento de confiabilidade, etc. Nesse sentido, os processos de um grupo interagem

entre si e com processos de outros grupos por meio de várias formas de comunicação que segundo BACON ( 1981 ), são as seguintes:

- comunicação um para um;
- comunicação vários para um;
- comunicação um para vários;
- comunicação vários para vários;

FRANK et alii, citado em KIRNER e MENDES ( 1988 ), abordou o problema de outra maneira, levando em consideração as possibilidades de emissão das mensagens, que podem ser:

- emissão única ( "unicast" )
- difusão ( "broadcast" )
- emissão múltipla ( "multcast" )

que cobre parcialmente a classificação citada.

Tais tipos de emissão de mensagens, são também abordadas por LIANG et alii ( 1990 ), que classifica a emissão única e a difusão como casos especiais da emissão múltipla. A preocupação principal foi em descrever os requisitos para o suporte para comunicação em grupos de processos, de acordo com os diferentes tipos de aplicações distribuídas.

## c) UTILIZAÇÃO DE "BUFFERS"

Ainda segundo KIRNER e MENDES ( 1988 ), na comunicação entre dois processos de um Sistema Distribuído que estiverem localizados em computadores diferentes, a mensagem pode passar pelo seguinte trajeto: ser preparada num "buffer" do processo origem; ser remetida ao "buffer" do núcleo associado a este processo; passar por vários "buffers" da rede ( subsistema de comunicação ); chegar ao "buffer" do núcleo associado ao processo destino; e finalmente ser colocada no "buffer" do processo destino. Alguns desses "buffers" poderão não existir, dependendo do tipo da rede e da comunicação utilizada.

Os "buffers" têm dupla função: servem para controlar situações de congestionamento na rede e, também, controlar o fluxo de informação.

O controle de congestionamento está relacionado com o subsistema de comunicação e, portanto, com os problemas de topologia da rede de comunicação, meio de transmissão, algoritmo de roteamento das mensagens, etc. Como se está preocupado com a comunicação entre processos, logo, com abstração dos problemas daquele nível de

comunicação, não se entrará nos detalhes a respeito. Para esclarecimento, veja KIRNER e MENDES ( 1988 ) e as referências ali citadas.

O controle de fluxo, por sua vez, deverá existir quando a emissão de mensagens precisar ser controlada para evitar que a chegada delas seja mais rápida do que a capacidade de sua manipulação por parte do receptor.

Assim, enquanto o controle de fluxo atua nos elementos terminais da comunicação ( processos ), o controle de congestionamento deve ocorrer na parte intermediária da comunicação ( subsistema de comunicação ). Muitas vezes, o controle de fluxo é utilizado para eliminar ou minimizar problemas de congestionamento.

Com relação ao controle de fluxo, segundo KIRNER e MENDES ( 1988 ), existem três possibilidades:

(1) dotar apenas o processo receptor de "buffers", em quantidade suficiente para absorver todas as informações a ele dirigidas;

(2) manter os "buffers" na origem sob controle do processo emissor e enviar primeiro um resumo, denominado de anúncio, da mensagem para o receptor. Quando esse for aceito, significará que o processo receptor está apto a receber a mensagem propriamente dita; e

(3) não se utiliza nenhum "buffer". O processo emite a mensagem estando o processo receptor pronto para recebê-la ou não. Caso não esteja, a mensagem é perdida e deverá ser retransmitida até um certo número de vezes.

Todas essas três formas apresentam suas deficiências que estão analisadas na referência mencionada, que resumindo, poderia-se dizer em termos gerais que a opção por alguma delas depende do tipo de aplicação.

Observa-se que, naturalmente, podem existir outras formas bastante distintas dessas, como também, algumas que sejam combinações delas.

## D) FALHAS NA COMUNICAÇÃO

Nos capítulos anteriores observou-se que um Sistema Distribuído deve tolerar faltas de "hardware" e de "software", a fim de permitir que ele continue operacional, na ocorrência de algumas delas, mesmo

que em sua forma degradada. Também, diante da preocupação na estruturação de sistemas em camadas, salientou-se que é importante que cada camada se preocupe em detectar o surgimento de algum evento indesejável no processamento normal de suas tarefas e tratá-lo, ou pelo menos detectar e sinalizar a ocorrência para que o "software" da camada superior possa tomar os devidos cuidados. Nesse sentido, o sistema de comunicação deve possuir um esquema de detecção de erros capaz de permitir que somente as mensagens corretas sejam encaminhadas aos seus destinos.

No caso de falhas na comunicação, muitas delas são previsíveis e bem determinadas e, portanto, podendo-se tratá-las de forma específica e direta ( tratamento de exceções ). Algumas, porém, consegue-se detectá-las após decorrido um tempo, o que pode dificultar os seus tratamentos da mesma forma que aquelas, exigindo-se um meio mais genérico, como o bloco de recuperação, o que pode implicar em tempo extra de processamento visto pelos usuários como não produtivo. Às vezes, como já mencionado, é melhor ou mais adequado, deixar o tratamento da falha para os processos que se utilizam do sistema de comunicação.

Diante da importância desse tópico para Sistemas Distribuídos, encontra-se na literatura uma boa quantidade de propostas de tipos de mecanismos de comunicação que contêm um certo grau de confiabilidade, com respeito ao tratamento de alguns tipos de falhas que porventura possam surgir, como por exemplo KOHLER ( 1981 ), GENTLEMAN ( 1981 ), etc. Como o item IV.3.1.3 abordou com uma certa profundidade tal assunto, aqui se fará apenas uma complementação com a descrição da falha lógica denominada de bloqueio perpétuo ( "deadlock" ), citada em KIRNER e MENDES ( 1988 ). O bloqueio perpétuo poderá resultar de um conjunto de processos que, de forma cíclica, estejam procurando enviar ou receber mensagens de um para outro. A solução desse problema costuma ser obtida por meio de análise de grafos que facilitam a visualização dos impasses.

### V.1.3 PORTOS

O conceito de porto como mecanismo de comunicação foi introduzido por BALZER e WALDEN ( 1971 ), citada em SEGRE ( 1987 ), como forma de identificação indireta na troca de mensagens entre

processos. Para isso, os portos devem representar canais de comunicação entre processos e devem mascarar as suas identidades. Assim, portos são entidades distintas de processos, podendo ser considerados como tipos abstratos de dados que são manipulados por um conjunto predefinido de operações ( primitivas ) e devem ser referenciados por meio de identificadores globais ou locais, no sentido de que eles sejam definidos a nível de sistema ou a nível de apenas algum de seus elementos ( por exemplo, um processo ( SEGRE, 1987 ) ), respectivamente.

Dessa forma, de um modo geral, quando um processo quer enviar uma mensagem a um outro processo, o faz emitindo-a a um porto de seu conhecimento ( por meio da primitiva envia, por exemplo ), o qual deve se encarregar de encaminhá-la ao processo destino que, por sua vez, para receber, deve solicitar ao porto ( através da primitiva recebe, por exemplo ) por onde tal mensagem deve chegar. Note, então, que um processo para se comunicar com um outro não obrigatoriamente necessita conhecer a identificação deste, valendo também a recíproca, ou seja, um processo que queira receber uma mensagem, não precisa explicitamente saber quem está enviando. É função do mecanismo de porto o estabelecimento do canal de comunicação entre os processos e, portanto, da tramitação de mensagens entre eles. Para tanto, tal mecanismo pode executar a referida função de forma implícita ou explícita. Na forma implícita, o estabelecimento do canal de comunicação poderia ser feito, por exemplo, em tempo da criação dos portos. Já na forma explícita, normalmente, o mecanismo deve oferecer primitivas específicas para isso, como por exemplo, primitivas para interligação de portos. Além disso, essas formas de estabelecimento de canais de comunicação, podem ser de dois tipos: estáticas, quando elas somente podem ser efetuadas na iniciação do sistema ( configuração estática ); ou dinâmicas, quando elas podem ser realizadas durante a execução do sistema ( configuração dinâmica ).

Como nos outros tipos de mecanismos de comunicação, as primitivas de comunicação podem ser síncronas ou assíncronas e, podem ou não se utilizarem de "buffers" para o armazenamento temporário de mensagens. Entretanto, a característica de comunicação, onde as identificações dos processos envolvidos não precisam ser explicitadas nas primitivas de comunicação, também denominada de endereçamento funcional, é que impõe a principal diferença dessa forma de

comunicação em relação às outras ( Vinter et alii, citado em KIRNER e MENDES, 1988 ).

De acordo com SEGRE ( 1987 ), pode-se associar um tipo de dados a um porto que identifica as mensagens que podem ser transmitidas por meio dele, fornecendo com isso, proteção em relação aos objetos comunicados. Essa característica, que também pode ser incluida nas outras formas de comunicação, permite que se evite troca de mensagens entre processos não compatíveis, como melhor se esclarecerá mais adiante. Além disso, possibilita que as mensagens sejam mais curtas, uma vez que não necessitam conter tal informação, o tipo do dado; o que representa uma certa economia no tempo de transmissão. Pode-se também associar a um porto uma lista definindo direitos de acesso ( que elementos do sistema pode acessá-lo ), pode-se permitir que os direitos de propriedade possam ser transferidos, e etc. ( LINDEN, 1976; DENNING, 1976; CORSINI et alii, 1984 ), enfim, associar vários mecanismos de proteção . Pode-se ainda, impor aspectos de tolerância a falhas de comunicação, por exemplo como aqueles vistos no item IV.3.1.3.

Os portos são classificados de acordo com as restrições a eles impostas. Quanto ao sentido da transferência da mensagem e quanto à forma de identificação.

Quanto ao sentido da transferência da mensagem, um porto pode ser de entrada, quando usado para receber mensagem; de saída, quando utilizado para enviar mensagem; e bidirecional ou de difusão, quando a mensagem pode fluir nos dois sentidos ( veja figura V.1 - KIRNER e MENDES, 1988 ).

Quanto à forma de identificação, um porto pode ser global ou local. Global, no sentido de que ele é conhecido a nível de sistema e, portanto, a princípio podendo ser utilizado por qualquer processo deste; dependendo da implementação, como se verá no item Interligação de Portos. Local, quando um porto deve ser particular a alguma entidade do sistema, por exemplo, um processo; ou seja, cada porto deverá ter o seu proprietário, que será o processo que o criar, que será o único com direitos para sua utilização como sua destruição, de acordo com Reid e Stemple et alii, citados em SEGRE ( 1987 ) e KIRNER e MENDES ( 1988 ).

Diante disso tudo, percebe-se que é possível implementar diferentes formas de comunicação e, portanto, diversas topologias, como melhor ficará claro a seguir.

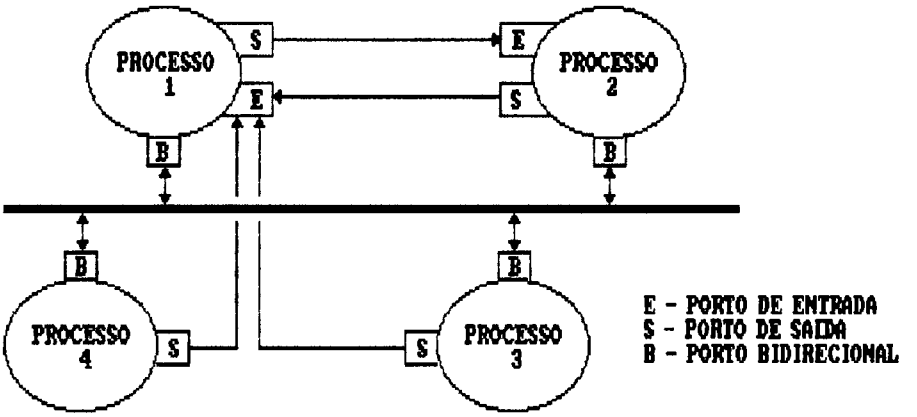



Figura V.1: Esquemas dos três tipos de portos locais

### V.1.3.1 INTERLIGAÇÃO DE PORTOS

Como visto, as primitivas de comunicação baseadas em portos não precisam referenciar diretamente as identificações dos processos que desejam trocar mensagens e, portanto, há a necessidade de um mecanismo para estabelecer os canais de comunicação definindo os enlaces a nível de processos. Dessa forma, pode-se ver os portos e processos como entidades distintas que, a princípio, facilita o emprego da programação modular ( DeREMER e KRON, 1976; GHEZZI e JAZAYERI, 1982 ). Tal técnica permite que um sistema possa ser programado como um conjunto de processos totalmente independentes que podem cooperar através de troca de mensagens, cujos enlaces de comunicação podem ser definidos a posteriori, estabelecendo a configuração desejada ao sistema ( KRAMER, 1983, 1984, 1985; SLOMAN, 1983, 1984; SEGRE, 1987; etc. ).

Os enlaces, ou estabelecimento dos canais de comunicação, como já mencionado, podem ser realizadas de forma implícita ou explícita e os portos podem ser globais ou locais.

A importância do estabelecimento dos enlaces de comunicação a nível de processos, está justamente no fato de que um porto, sozinho, é uma entidade que no máximo seria capaz de reter as informações a ele enviadas. Assim, de um modo geral, um porto sempre está associado a um processo ( aquele que o criar ), sendo ele global ou local.

Desse modo, antes de descrever o enlace implícito, analisar-se-ão os significados de porto global e local. Um porto global, como já dito, é

aquele cuja identificação é de conhecimento de todos os elementos do sistema, portanto, qualquer um pode acessá-lo ( a menos que exista uma certa proteção a respeito de direitos de acesso ). Já um porto local, sempre será de conhecimento de apenas do processo que o criou; podendo também ter características de proteção. Como se sabe, um porto local pode ser de entrada, ou de saída, ou bidirecional ( ou de difusão ). Assim, em termos de portos locais tem sentido dizer que um porto de saída de um processo está conectado ( estabelecido o canal de comunicação ou o enlace ) a um porto de entrada de um outro processo; o primeiro emite uma mensagem pelo seu porto de saída que se encarregará de encaminhá-la ao porto de entrada do outro processo, através do enlace estabelecido, que a receberá, por meio deste. Porém, em termos de portos globais, estabelecer a mesma forma de comunicação, é no mínimo de uma certa complexidade, pois como eles são de conhecimento de quaisquer processos do sistema, qualquer um poderia utilizar o porto de saída de um outro para poder enviar sua mensagem, assim, podendo estabelecer uma forma de comunicação nada elegante, exigindo uma atenção bastante grande dos programadores de sistema ( veja figura V.2 ). Uma forma para se impedir que se estabeleça uma conexão indevida, pelo menos em termos estruturais para não deixar o sistema amorfo, seria a colocação de mecanismo de proteção baseado em capacidade, por exemplo. Note, entretanto, que isso praticamente levaria ao esquema do emprego de portos locais. Percebe-se que isso acontece ao se usar portos globais de saída, aqueles que enviam mensagens, pois, eles exigem que exista a conexão com um porto de entrada para poder encaminhar as mensagens; senão o esquema norteante do mecanismo de comunicação baseado em portos é quebrado. Conseqüentemente, o uso de portos globais fica restrito a portos de entrada, resultando o seguinte esquema de comunicação: os processos que desejarem enviar mensagens a um outro, devem fazê-la diretamente ao porto de entrada desse ( veja figura V.3 ).

Diante disso, nota-se que quando se usa portos globais, as conexões ficam implicitamente estabelecidas. Porisso, apesar de ainda facilitar a programação modular, eles acabam limitando-a, já que cada módulo ( conjunto de processos ) não pode ser programado independentemente dos outros e compromete a flexibilidade do sistema ( SEGRE, 1987 ).

Por outro lado, quando se utiliza portos locais, diante da restrita visibilidade de suas identificações, há a necessidade da existência de

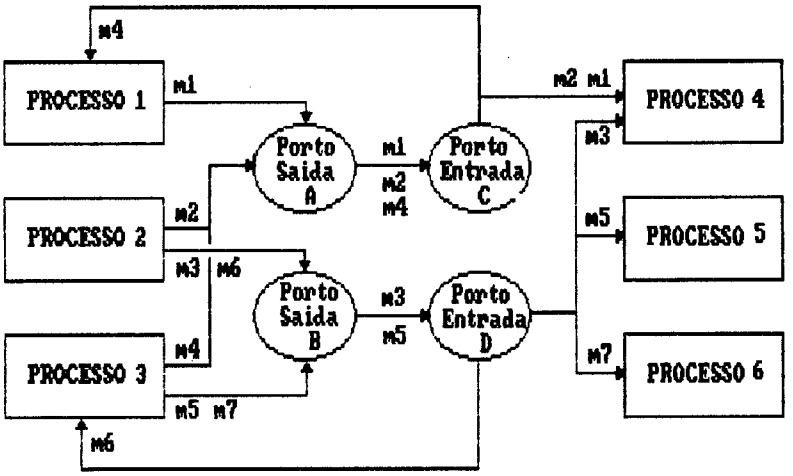



Figura V.2: Esquema de Portos Globais de Entrada e de Saída

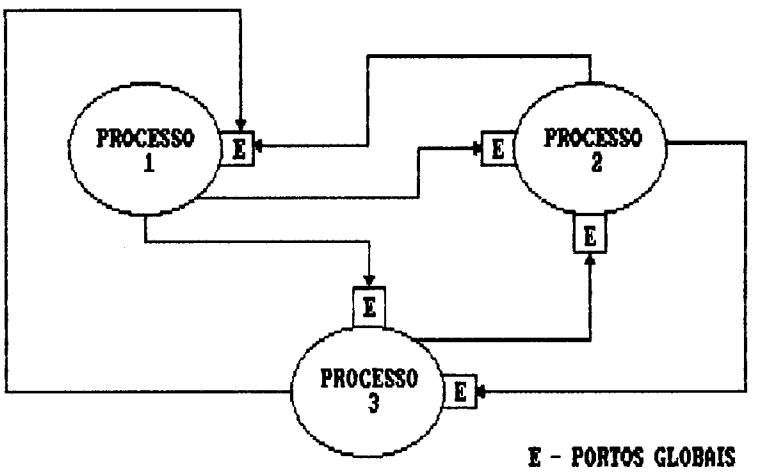



Figura V.3: Esquema de Portos Globais de Entrada

mecanismos auxiliares que permitam o estabelecimento das devidas conexões, as quais podem ser feitas de forma implícita ou explícita. Porém, de imediato, percebe-se que a forma implícita, além de acabar levando às restrições de dependência e flexibilidade acima observadas, ela em nada contribui na melhoria da outra forma de se estabelecer as conexões, como se notará a seguir.

De acordo com KIRNER e MENDES ( 1988 ), a conexão ou interligação de portos, pode dar-se de quatro maneiras: um para um; um para vários; vários para um; e vários para vários.

A conexão um para um, interliga um porto de entrada a um porto de saída, isto é, associa um emissor a um receptor.

A conexão um para vários, associa um conjunto de receptores com um único emissor. Esse esquema de comunicação, dependente de particularidades de implementação, permite o seguinte:

a) **conexão de difusão:** onde a mensagem que for enviada deverá chegar a cada um dos receptores;
b) **conexões múltiplas de leitura:** onde a mensagem enviada só chegará ao primeiro receptor que tentar recebê-la.

A conexão vários para um, associa vários emissores a um único receptor. Encontram-se aqui os seguintes esquemas:

- **conexões múltiplas de escrita:** onde o receptor deverá receber mensagem de qualquer emissor a ele ligado através do porto;
- **conexão de concentração:** onde a mensagem que o receptor recebe deverá ser a concatenação de cada mensagem enviada de cada emissor ligado a ele. Dessa forma a recepção só estará completada quando todos os emissores tiverem enviado mensagem.

A conexão vários para vários, associa um conjunto de emissores a um conjunto de receptores. Esse tipo de conexão pode ser obtido por meio da combinação das conexões um para vários e vários para um, gerando quatro possibilidades a seguir descritas.

1) **conexões múltiplas de escrita/conexão de difusão:** transmite cada mensagem para todos os receptores do conjunto;
2) **conexões múltiplas de escrita/conexão múltiplas de leitura:** transmite cada mensagem para o primeiro receptor que quiser recebê-la;
3) **conexões de concentração/conexão de difusão:** transmite a concatenação das mensagens de todos os emissores do conjunto para todos os receptores do conjunto; e
4) **conexões de concentração/conexão múltipla de leitura:** transmite a concatenação de todos os emissores para o primeiro receptor que quiser recebê-la.

### V.1.3.2 UTILIZAÇÃO DE "BUFFERS"

Como nos outros tipos de mecanismos de comunicação baseados em troca de mensagens, este baseado no uso de portos, também pode utilizar "buffers", tanto para resolver problemas de congestionamento a

nível do subsistema de comunicação, como para controlar o fluxo de informação.

As técnicas de utilização de "buffers" para o controle de congestionamento no subsistema de comunicação são as mesmas já descritas. Apesar de, praticamente, valerem as mesmas sugestões para o controle de fluxo, devida às pequenas particularidades e o interesse direto nesse tipo de mecanismo no presente trabalho, se aborda novamente esse assunto. É claro, agora com as preocupações restritas aos mecanismos de comunicação baseado em portos.

De um modo geral, os "buffers" podem ser alocados nos seguintes elementos de um sistema:

a) somente nos portos;
b) somente nos processos;
c) somente no sistema;
d) nos portos e nos processos;
e) nos portos e no sistema;
f) nos processos e no sistema; e
g) nos portos, no sistema e nos processos.

Não existe uma regra bem definida que imponha qual dessas possibilidades é a que realmente permite implementar a melhor forma para controlar o fluxo de informação. Ela depende de vários fatores, sendo alguns deles:

a) tipo de porto: global ou local;
b) tipo de conexão;
c) tipo de sincronização;
d) independência de velocidade de processamento dos emissores e respectivos receptores;
e) tolerância a falhas;
g) proteção;

Além disso, existem os problemas já citados, quanto à quantidade de "buffers" a alocar. Algumas propostas estão sugeridas em KIRNER e MENDES ( 1988 ).

Sem entrar em todos os detalhes, sugerem-se algumas formas para controlar o fluxo de informação, considerando os fatores mencionados.

No caso de portos globais, como justificado, tem sentido apenas portos de entrada. Os processos emissores se utilizam de primitivas do tipo envia para encaminhar as mensagens para um porto de entrada que, obrigatoriamente, deve pertencer a um processo que quer receber

mensagens, que o faz através de uma primitiva do tipo recebe dirigida a esse porto seu. Diante dessa restrição natural, parece que a única forma possível de comunicação é do tipo produtores x consumidores. Naturalmente, um consumidor pode ser produtor para um outro processo, o qual pode ser um de seus produtores ( veja figura V.3 ). Entretanto, como não há nada proibindo que se definam vários portos globais com características diferentes, por exemplo, porto com tipo associado, porto com proteção quanto aos processos que podem acessá-lo, etc, como também, não existe restrição quanto à quantidade de portos que um processo pode ter, abre-se todo o leque de opções descritas no item sobre interligações de portos. Evidentemente, de forma estática, como já observado.

Apenas para exemplificar, descreve-se uma forma de controle de fluxo de porto global que aceita qualquer tipo de mensagem e que permite que qualquer processo envie mensagem para ele e as respectivas primitivas dos tipos envia e recebe síncronos, e discute-se o caso quando a primitiva envia é não bloqueante, permanecendo a recepção bloqueante ( comunicação assíncrona ).

Quando a comunicação é síncrona, não haveria, a princípio, necessidade de se usar "buffers". Porém, como a comunicação pode ser de vários para um, portanto podem existir mais do que um processo querendo enviar mensagens a esse porto, é interessante utilizar "buffers" no porto, ou no núcleo do nodo que contém o processo receptor ( no sistema ); isso permitiria um certo ganho de desempenho, principalmente se os emissores forem remotos, pois eles ficariam suspensos apenas à espera dos sinais de que as mensagens foram recebidas e assim poderem prosseguir. Isso admitindo que o subsistema de comunicação é totalmente confiável, garantindo que as mensagens sempre chegam corretamente em seus destinos. Agora, se a primitiva envia é não bloqueante, poderia-se associar "buffers", também, no núcleo do nodo do processo emissor. Isso permitiria uma liberação mais rápida desse, pois bastaria a primitiva envia encaminhar a mensagem a esses "buffers", deixando para o envia de mais baixo nível transmití-la aos "buffers" do porto destino ( ou aos do núcleo do nodo destino ) - implementação de primitivas em uma hierarquia de camadas de operações, cada uma de mais baixo nível, até chegar ao nível do subsistema de comunicação. Como alertado, essas são algumas sugestões, descritas sem a preocupação em se definir a quantidade de "buffers" ( veja item D deste capítulo ), como também outras características que

porventura seriam interessantes diante dos tipos de aplicações que se deseja abranger, a fim de facilitar a programação, em termos das necessidades de cooperação entre processos ( veja KIRNER e MENDES, 1988 ).

Para o caso de portos locais, esquemas análogos podem ser aplicados. Apenas se deve tomar os devidos cuidados quanto às questões que diferenciam aqueles tipos de portos desses: visibilidade das identificações; e tipos de portos. Quanto à visibilidade, por as identificações desses não serem globais, ou mais especificamente, não sendo "tolerado" o acesso por qualquer um dos processos, salvo os seus proprietários, permite-se que estes sejam autônomos, como se fossem procedimentos de biblioteca. Quanto aos tipos de portos, enquanto que naqueles, praticamente, existe apenas porto de entrada, nesses podem-se criar portos unidirecionais de entrada ou de saída, ou portos bidirecionais ( ou de difusão ) e, assim, necessitando que eles sejam interligados para o estabelecimento dos canais de comunicação e definido, dessa forma, a topologia da intercomunicação entre os processos. Conseqüentemente, agora, por exemplo, um porto de saída é quem sabe para quais portos de entrada ou bidirecional, as mensagens a ele dirigidas devem ser enviadas. Isso acaba influindo na escolha da localização dos "buffers", tanto para melhorar o desempenho do próprio mecanismo de comunicação, como também, na estruturação das primitivas de comunicação e sincronização. A palavra tolerado, acima utilizada, foi frisada, porque esses portos deverão de uma forma ou outra ser de conhecimento do sistema de configuração, pois senão este não poderá estabelecer as devidas conexões ( configuração estática ou dinâmica ).

### V.1.3.3 OPERAÇÕES NECESSÁRIAS AO COMPORTAMENTO DINÂMICO DO SISTEMA

Um sistema apresenta um comportamento dinâmico quando seus elementos sofrem variações como criação e destruição de processos, criação e destruição de portos, conexões e desconexões de portos, etc. ( KRAMER et alii, 1985; LOPES, 1988; KIRNER e MENDES, 1988 ). Tais variações podem ser necessárias que ocorram para solucionar problemas de sobrecarga, ou como conseqüência de tratamento de falhas, ou para permitir seu crescimento incremental, ou etc. ( veja item IV.4 Migração de Processos ).

De acordo com Reid citado em KIRNER e MENDES ( 1988 ), as operações envolvidas com essas variações são as seguintes:

- **criação de processos:** passará a existir um novo processo que começará a executar logo que for criado;
- **destruição de processo:** o processo deixará de existir como parte do sistema;
- **criação de porto:** passará a existir um novo porto, pertencente ao processo que o criar, associado com um identificador definido na operação de criação;
- **destruição de porto:** o porto deixará de existir como parte do sistema. Essa operação só poderá ser executada pelo processo dono do porto;
- **conexão de portos:** será estabelecida a conexão de comunicação entre dois portos. Os portos só poderão ser conectados se forem compatíveis e de sentidos opostos. Um porto pode estar ligado a muitos portos ao mesmo tempo e a tentativa de conexão de porto já ligado não terá nenhum efeito; e
- **desconexão de portos:** será desfeita a conexão de comunicação entre dois portos. Essa operação poderá ser executada por qualquer processo que conhecer os dois portos envolvidos.

Essas operações não devem ser executadas livremente, elas devem satisfazer algumas restrições em sua seqüência, que são:

- um processo possuidor de um porto deverá ser criado antes da criação do porto;
- os portos deverão ser criados antes do estabelecimento de suas conexões;
- um porto deverá estar completamente desconectado antes de ser destruído;
- o porto deverá ser destruído antes de se destruir o processo dono do porto;
- um processo pode ser destruído se não contiver nenhum porto;
- as conexões deverão ser realizadas antes da aplicação de primitivas de comunicação sobre eles;

## V.2 SISTEMAS OPERACIONAIS DISTRIBUÍDOS

KIRNER ( 1986 ) e KIRNER e MENDES ( 1988 ), fazem uma análise sucinta sobre alguns tipos de Sistemas Operacionais Distribuídos encontrados na literatura, focalizando basicamente a arquitetura de

"hardware", estruturação do "software" e as linguagens utilizadas e citando os objetivos que os nortearam. Já em SEGRE ( 1987 ), as referências citadas em termos de Sistemas Operacionais Distribuídos, foram com a preocupação principal em mostrar as vantagens da utilização de determinados tipos de linguagens de alto nível para suas implementações; isso devido aos seus objetivos estarem concentradas sobre linguagens de programação, quanto às suas facilidades para programação de sistemas, as LPP ( Linguagens de Programação de Pequena escala ) e LPL ( Linguagens de Programação de Larga escala ).

Diante disso, considerando as referências citadas nessas duas acima mencionadas, e tendo em vista que os objetivos desse trabalho difere em muito daqueles, procura-se agora descrever de forma resumida alguns tipos de Sistemas Operacionais Distribuídos que foram elaborados com alguma preocupação a respeito de tolerância a falhas, mais especificamente, com respeito a facilidades para migração de processos; a qual não necessariamente, como já visto, é desejada apenas para impor tolerância a falhas, pode ser para permitir o balanceamento dinâmico de carga, tanto no sentido de melhor distribuir os processos pelos processadores alocando-os em suas memórias, como também para aliviar o congestionamento da rede física de comunicação, procurando colocar os processos com maior interação para serem executados por um mesmo processador. Para tanto, o tipo de mecanismo de comunicação tem um papel fundamental, assim, procurar-se-á também dar uma visão. Não se pretende entrar em detalhes por não se considerar de vital importância para o presente trabalho, o objetivo é de apenas historiar as épocas em que efetivamente se iniciou a construção de Sistemas Operacionais Distribuídos com alguma preocupação a respeito desses aspectos. Além disso, para servir como uma referência bibliográfica, ou melhor, uma bibliografia comentada, principalmente para os capítulos VI e VII ( Desenvolvimento de um Suporte para Construção de Sistemas Distribuídos Tolerantes a Falhas e Projeto de um Suporte para Sistemas Operacional Distribuído com Reconfiguração dinâmica de Processos ). Dessa forma alguns tipos de Sistemas Operacionais analisados naquelas referências, serão novamente aqui resumidas, diante dos interesses descritos.

## SISTEMA OPERACIONAL DISTRIBUÍDO ROSCOE

Roscoe ( SOLOMON e FINKEL, 1979 ) é um Sistema Operacional implementado na Universidade de Wisconsin que permite que uma rede de microcomputadores cooperem para prover uma facilidade de computação de uso geral.

Quanto aos aspectos de tolerância a falhas, os autores apenas citam a possibilidade de tolerar defeitos em processadores e suas reintegrações, indicando quando isso seria permitido diante da estruturação do Sistema Operacional implementado.

Quanto à possibilidade do Roscoe permitir a migração de processos, os autores deixaram, a priori, para posterior estudo. Assim, apenas descrevem os problemas que perceberam para que o gerente de recursos contenha essa facilidade. Os problemas citados foram: a translação dinâmica de endereços, pois, apesar de os processadores serem idênticos no caso, nem sempre o código de um processo migrado poderá ocupar o mesmo espaço de memória da origem; a função do roteador de mensagens deverá ser capaz de descobrir a nova localização do processo. Para esses problemas, os autores descrevem algumas soluções consideradas teoricamente possíveis. Alertam, entretanto, para um problema considerado fundamental que é: como decidir quando e para onde mover um processo, diante da possibilidade de se criar uma sobrecarga no meio físico de comunicação. Apesar de lembrarem de algumas heurísticas, eles julgam que o problema pode ser resolvido apenas por experimentação.

Quanto ao mecanismo de comunicação utilizado, o Roscoe emprega um tipo semelhante ao de porto global, que denominaram de enlace. Tal mecanismo combina os conceitos de caminho de comunicação e capacidades. Cada enlace representa uma conexão lógica unidirecional entre dois processos: o possuidor que pode enviar mensagens pelo enlace e o proprietário, que recebe-as. O possuidor pode duplicar o enlace ou dá-lo a um outro processo, sujeito às restrições impostas pelo próprio enlace, enquanto que o proprietário, o criador do enlace, não pode fazer nenhuma alteração, apenas especificar certas propriedades, por exemplo, se o enlace pode ou não ser copiado, que ele pode ser usado apenas uma vez, ou sua destruição enviará uma notificação ao seu proprietário.

O mecanismo implementado a nível de núcleo, utiliza "buffers". Cada núcleo mantém uma quantidade limitada de "buffers" que são

alocadas entre todos os processos locais. Mensagens que chegaram e que ainda não foram recebidas são enfileiradas nesses "buffers", como também as mensagens que deverão ser enviadas para os processadores vizinhos.

## MICROS: UM SISTEMA OPERACIONAL DISTRIBUÍDO PARA MICRONET, UMA REDE DE COMPUTADORES RECONFIGURÁVEL

Segundo os autores, WITTIE e TILBORG ( 1980 ), o projeto do "hardware" do Micronet teve inicio em 1976 com o objetivo de servir como uma bancada para experimentações com diferentes arquitetura de rede de computadores com Sistemas Operacionais Distribuídos como o Micros, e com programas para aplicações paralelas para resolver problemas nos campos como análise numérica, inteligência artificial e simulação de rede.

O projeto do Sistema Operacional Distribuído Micros, iniciado em 1978, implementado em Pascal Concorrente, inclusive o seu núcleo, foi norteado para desenvolver princípios de organização que pudessem ser estendidos para controlar redes de computadores com milhares de nodos. As outras metas foram:

- para suportar um ambiente multiusuário com variação dinâmica de programas grandes e pequenos de usuários. A rede deve ser capaz de manipular variação de carga de milhares de usuários realizando pequenas tarefas que podem ser executadas em um nodo, como executar um único programa gigantesco que exigirá todos os nodos para o seu processamento.
- diante da possibilidade da ocorrência de defeitos nos nodos e nos enlaces de interconexão, o sistema Micros deverá tolerar falhas em regiões locais. Falhas deverão causar, no máximo, perda de informações locais do sistema e de programas locais de usuários. Defeitos de componentes deverão ser detectados, removidos, reparados e recolocados enquanto o resto da rede deverá continuar servindo pedidos de usuários.
- o Sistema Operacional projetado não deve depender muito do padrão de interconexão ou topologia. Micros deverá ser projetado para funcionar razoavelmente bem em redes de qualquer topologia, pelos seguintes motivos: a) diante da possibilidade da ocorrência de falhas locais de "hardware" que poderão eliminar nodos e linhas de comunicação, assim, Micros deverá dinamicamente acomodar pequenas

mudanças locais da topologia da rede, mesmo que esta seja supostamente fixa; b) se a rede de comunicação for aderente à aplicação, isso poderá implicar na necessidade de se escrever um Sistema Operacional para cada tipo de rede, o que seria um grande desperdício de tempo.

- Micros se adapta a mudanças na carga de comunicação. Especialmente em uma rede para multiusuários processando uma mistura de tarefas, variando desde poucas a muitas tarefas cooperando, a densidade do tráfico de mensagens nos diferentes caminhos de comunicação variará muito e provavelmente poderá ser inaceitável. Considerando que o componente de maior custo do "hardware" da rede de computadores será o da comunicação internodos, Micros deve otimizar a sua utilização. Em particular, ele deve detectar e eliminar dinamicamente gargalos de comunicação. Para realizar isso, Micros deve conter mecanismos para monitorar os fluxos de mensagens. Possìvelmente terá que migrar ou replicar processos ou banco de dados dentro da rede para aliviar a sobrecarga nas linhas de comunicação.

O projeto Micros, nessa época, não foi implementado atendendo todos esses objetivos. Implementou-se uma rede com 16 microcomputadores LSI 11/23 e um Sistema Operacional básico, não contendo as características para tolerância a falhas delineadas nos objetivos. Entretanto, diante do interessante esquema de estruturação do Sistema Operacional Distribuído para ser independente do "hardware" da rede e que deve prover alguma tolerância a suas falhas, procura-se a seguir, dar uma breve descrição.

O Micros foi logicamente estruturado adotando-se uma hierarquia muito semelhante à de uma corporação militar ou civil de grande escala. Isto é, é uma hierarquia em árvore truncada, onde os nodos de mais baixo nível ( as folhas ), executam as tarefas dos usuários e manipulam os dispositivos de entradas e de saídas conectados à rede. Os nodos superiores, gerenciam os recursos e provêem o controle regional dos nodos logo abaixo ( os recursos ). Para tanto, os recursos são globalmente acessíveis, entretanto, eles são gerenciados na forma de conjuntos aninhados. Cada nodo gerente controla, possivelmente de forma indireta, todos os recursos de uma sub-árvore na hierarquia da rede. A monitoração, escalação e alocação de cada recurso associado com o nodo mais baixo são executadas por um ou mais desses nodos gerentes na cadeia entre o nodo recurso e o topo da hierarquia. Para

evitar sobrecarga de processamento e de comunicação, cada nodo da árvore troca diretamente informações de controle somente com os que estão no nível logo acima, seus mestres, e com os que estão logo abaixo, seus escravos. Entretanto, qualquer nodo pode fisicamente estar conectado a muitos outros em vários níveis. Dessa forma, tarefas de usuários e mensagens de dados de entrada/saída, podem estar sendo substituídos por qualquer um no enlace físico, sem a preocupação da estrutura lógica de controle hierárquico. Em outras palavras, a sub-rede de comunicação não está ciente das distinções funcionais dos processadores hospedeiros. Também para evitar sobrecarga, os nodos os nodos de cada nível mantém somente o resumo da informação sobre os recursos do nível logo abaixo. Assim, no ponto mais alto na hierarquia da árvore sabe-se de tudo, porém, com poucos detalhes, enquanto que nos pontos mais baixos, sabem-se dos detalhes de cada região local.

Nessa estruturação, caso o gerenciamento de nível mais alto fosse executado por um único nodo, o sistema todo ficaria vulnerável às falhas deste. Conseqüentemente, esse gerenciamento é feito por vários nodos formando uma oligarquia para controle global. Cada um dos membros da oligarquia regularmente passa um sumário de informações para os outros membros, para justamente se protegerem das falhas de "hardware". Similarmente, os gerentes de nível mais baixo passam sumários para os seus gerentes de nível imediatamente superior, inclusive as listas de seus escravos. Desse modo, mesmo que um nodo entre em colapso, o mestre logo acima pode se comunicar com os escravos daquele. Juntos eles contêm informação suficiente para selecionar um novo gerente e reconstruir os dados para gerenciamento dos recursos, especialmente já que todos os dados exceto aqueles sobre as recentes translações de controle estão mantidos em detalhes pelos escravos. As mensagens para um gerente destruído, não serão apropriadamente conhecidas, fazendo com que as transações incompletas sejam abortadas. Pedidos repetidos deverão fazer com que o novo gerente inicie transações equivalentes. O único efeito direto decorrente do colapso de um nodo gerente sobre as tarefas de usuários deverá ser a perda de pacotes de mensagens de usuários que estavam mantidos no nodo que apresentou defeito. Agora, se se utilizar protocolos fim a fim, os pacotes perdidos poderão ser novamente emitidos. No nível mais baixo da hierarquia da árvore, o colapso de um nodo implicará na perda de tarefas de usuários, a menos que elas sejam replicadas em outros nodos.

Quanto ao tipo de mecanismo de comunicação e sincronização, Micros utiliza o da caixa postal, cada uma mantido por um monitor, como descrito por HANSEN ( 1977 ), motivado provavelmente por se ter utilizado a linguagem Pascal Concorrente.

## SODS/OS: UM SISTEMA OPERACIONAL DISTRIBUÍDO PARA IBM SÉRIES/1

O objetivo principal dos autores SINCOSKIE e FARBER ( 1980 ), foi o de produzir um sistema descentralizado com controle descentralizado. Para isso, todos os processos do sistema SODS/OS são independentes de locação. Cada processo pode estar em execução em uma máquina, se comunicar com processos residentes em outra máquina e estar executando entrada/saída em uma terceira máquina, sem absoluto conhecimento de qualquer separação física. Além disso, SODS/OS poderia causar migração de vários processos de uma máquina para uma outra, sem que eles fiquem cientes do movimento.

No SODS/OS os processos se comunicam por meio de caixas postais. Uma caixa postal é uma fila FIFO ( "first-in first-out ) de mensagens residente fora do espaço de endereçamento de qualquer processo. Um processo envia ou recebe uma mensagem de caixas postais. Quando um processo requisita a recepção de uma mensagem, ela é primeiro retirada da fila e em seguida copiada no espaço de endereçamento do processo. Tal mecanismo está sujeito às seguintes restrições: se um processo tentar receber uma mensagem de uma caixa postal vazia, ele será bloqueado pelo sistema até que um outro processo envie uma mensagem para essa caixa postal; analogamente, se um processo tentar enviar uma mensagem para uma caixa postal cheia, ele será bloqueado até que um outro processo retire uma dessa caixa postal.

Apesar de se ter citado a migração de processos como uma possível facilidade do SODS/OS, ela não foi sequer tocada.

## XOS: SISTEMA OPERACIONAL PARA ARQUITETURA X-TREE

MILLER e PRESOTO ( 1981 ), interessados em investigar os efeitos de uma arquitetura X-TREE sobre o projeto de Sistema Operacional para uso geral, desenvolveram o XOS, também, com vistas aos seguintes objetivos:

- compartilhamento de recursos: desde que o sistema é para ser do tipo distribuído, ele tem que permitir que os processos residentes em qualquer lugar da árvore, compartilhem informações;
- uso efetivo da árvore de conexão: a principal razão para muitas das decisões arquiteturais em X-TREE foi para reduzir o tráfego;
- migração de processos: como um passo em direção à redução de tráfego provocado pelos processos dos usuários, adicionou-se os requisitos que os processos sejam capazes de migrarem através do sistema para os nodos que poderiam minimizar o tráfego na árvore.

Os autores naquela época apenas chegaram a descrever como poderia-se migrar processos, acreditando, para atender o objetivo acima, que as técnicas de estruturação utilizada facilitaria tal tarefa. Observa um problema decorrente de uma migração, que é sobre as mensagens que estão em trânsito pela árvore e, portanto, não cientes da nova locação do processo destino. Apresentam a retransmissão das mensagens como uma solução, após a identificação da nova locação.

Quanto ao mecanismo de comunicação entre processos, foi utilizado o mesmo para o sistema DEMOS ( BASKETT et alii, 1977 ), ou seja baseado no conceito de portos, mais precisamente, portos locais com capacidades.

## SISTEMA OPERACIONAL DISTRIBUÍDO AMOEBA

O Sistema Operacional Distribuído AMOEBA, discutido por TANENBAUM e MULLENDER ( 1981 ), é um sistema que intenciona controlar uma coleção de máquinas baseadas na idéia de um "pool" de processadores. O modelo utilizado para comunicação entre processos é o serviço. Um serviço é definido por um conjunto de comandos e respostas, no estilo de tipo abstrato de dados. É implementado por um ou mais processos servidores que aceitam mensagens e executam o trabalho solicitado. A quantidade de servidores por serviço é determinado pelo provedor do serviço e não deverá ser visível aos seus usuários.

Internamente, Amoeba faz distinção entre serviços públicos e privados. Os serviços públicos são aqueles que sempre estão esperando por novos pedidos de execução de seus serviços e porisso, são de longa vida. Tipicamente, são serviços de arquivo ( leitura e escrita de arquivos ), serviços de diretoria ( nomeação de arquivos e gerenciamento de arquivo ), serviço de base de dados, etc. Já os

serviços privados, são de vida curta, iniciados apenas para executar específicos programas para um específico usuário. Por exemplo, um usuário pode solicitar ao sistema que execute seu programa cujos dados devem ser obtidos de um arquivo e os resultados armazenados em um outro, no estilo de comando "pipeline" do UNIX.

Cada serviço deve conter um ou mais portos, por meio dos quais se pode acessar os seus préstimos. Tais portos são protegidos pelas suas identificações, no caso números. Quem conhece o porto de um serviço tem o direito de acessá-lo. Assim, para os serviços públicos, os portos são globais e para os privados, os portos utilizados devem ter suas identificações mantidas em segredo, ou seja, devem ser de conhecimento restrito a alguns tipos de processos. A fim de evitar que portos distintos recebam uma mesma identificação, o sistema se responsabiliza por tal tarefa.

Quanto à migração de processos em Amoeba, é apenas citado tal facilidade a qual deve ser executada pela camada denominada de monitor do protocolo de comunicação, apenas para solucionar o problema de balanceamento dinâmico de carga. Os autores não chegam sequer a citar os problemas decorrentes de uma migração, se limitam apenas a descrever que quando uma máquina deseja se livrar de um processo, ele simplesmente, por meio do monitor, envia, através de um porto criado especialmente para isso, o descritor do processo para uma máquina que queira recebê-lo. Quando isso acontecer, o monitor envia o código do processo e, em seguida, elimina o porto e o próprio processo.

## LOCUS: UM SISTEMA DISTRIBUÍDO ALTAMENTE CONFIÁVEL E TRANSPARENTE À REDE

Segundo POPEK et alii ( 1981 ) e POPEK e WALKER ( 1985 ), LOCUS é um Sistema Distribuído altamente confiável e transparente a rede, enquanto que ao mesmo tempo sustenta alto desempenho e replicação automática de recursos a um grau indicada pelos perfis de confiabilidade associadas. Por transparência de rede, se quer dizer que o conhecimento da rede e do código para interagir com as estações ficam escondidos tanto dos usuários como dos programas sob condições normais de utilização. O LOCUS é compatível com o sistema UNIX, com desempenho comparável em relação a um único sistema UNIX. É executado sobre uma rede local de baixo retardo, de grande largura de banda de transmissão e de baixa taxa de erros, foi projetado para

permitir que cada estação tenha um significativo alto grau de autonomia e a nível de rede, uma estrutura de nome independente de locação.

Como citado, LOCUS procura impor confiabilidade por meio de redundância de recursos. Foram tomadas quatro grandes passos para encontrar o potencial para confiabilidade e disponibilidade interessantes que estejam presentes em Sistemas Distribuídos.

Primeiro, impor operação confiável por meio da habilidade de poder substituir versões alternativas de recursos, quando o original falhar.

Segundo, utilização de operações sobre arquivos no esquema do protocolo de comprometimento; ou seja, operações atômicas sobre arquivos.

Terceiro, uma estação se for desconectada da rede, ela pode ainda prosseguir com os seus trabalhos locais; autonomia.

Quarto, a interação entre máquinas é fortemente estilizada para constituir uma cooperação bastante protegida. Cada máquina faz uma extensa verificação de consistência defensiva da informação.

## SISTEMA OPERACIONAL DEMOS/MP

O Sistema Operacional DEMOS/MP, descrito por POWELL e MILLER ( 1983 ), é uma versão do Sistema Operacional DEMOS discutida por BASKETT e MONTAGUE ( 1977 ), estendida para operar em um ambiente distribuído. Nesse sistema a facilidade da migração de processos foi abordado com profundidade, apresentado como solução para se obter uma boa distribuição da carga de trabalho entre os processadores e como mecanismo suporte para recuperação de falhas. Tudo isso devendo ser realizado de forma invisível aos usuários.

Para isso, DEMOS/MP é um Sistema Operacional baseado em mensagens, com a comunicação se constituindo o seu maior mecanismo básico. Em cada processador do Sistema Distribuído ( cada hospedeiro ), reside uma cópia de seu núcleo que é orientado à comunicação; embora cada núcleo mantenha seus próprios recursos, eles cooperam para prover a facilidade da troca de mensagens entre processos independente de locação destes e de forma confiável. Assim, todas as interações entre um processo e um outro ou entre um processo e o sistema são feitas via chamadas do núcleo.

DEMOS/MP usa enlace ( "link" ), como mecanismo para comunicação. Um enlace é um ponteiro que especifica o receptor da

mensagem. Tal endereço é globalmente protegido e acessado via o espaço de nome local, o processo que o cria é o seu dono.

Enlaces podem ser criados, duplicados, passados para outros processos, ou destruídos e são manipulados como capacidades; isto é, o núcleo participa de todas as operações de enlace, porém o controle conceitual de um enlace é empossado pelo processo para quem ele aponta ( o seu criador ). Eles são independentes do contexto, de forma que quando passados para processos diferentes, continuam apontando para os seus criadores.

O enlace possui um atributo especial denominado de Transferir_Par_Núcleo, por meio do qual permite que o núcleo transmita mensagens para processos residentes em processadores distintos, indiferente se o processo destino ainda reside ali ou está em movimento ( em migração ). Caso o processo destino esteja em movimento, o núcleo deve armazenar a mensagem para posterior transmissão quando aquele estiver pronto para receber. Além disso, esse atributo permite que o núcleo aceite pedidos do gerente de processos para controlar algumas funções como suspender a execução de um processo, estabelecer canal de leitura ou de escrita direta na memória de um processador qualquer do sistema, etc.

Esse mecanismo, segundo os autores, simplifica bastante os problemas associados com o movimento de processos, como se perceberá a seguir.

Em DEMOS/MP, a migração de um processo segue os seguintes passos:

1) remover o processo em execução: o processo é simplesmente marcado como em migração;
2) solicitar o núcleo destino para mover o processo: é enviado uma mensagem para o núcleo do processador destino, solicitando-o para migrar o processo para sua máquina. Essa mensagem contém informações sobre o tamanho e locação do estado residente do processo, código, etc.;
3) alocar um estado de processo no processador destino: é criado um estado de processo vazio. Esse estado é semelhante àquele alocado quando da criação do processo, exceto que ele tem o mesmo identificador do processo em migração. Recursos tais como espaço de permuta de memória virtual é feita nesse instante;

4) transferir o estado do processo: é feito por meio da facilidade de mover dados do núcleo destino, que copia o estado do processo em migração para o estado vazio;
5) transferir o programa: de forma análoga ao passo anterior, o núcleo copia a memória ( código, dados e pilha ) do processo para o processo destino, cujo espaço deve ser reservado pelo próprio núcleo destino. Após isso, o controle é retornado para o núcleo fonte, isto é, para o núcleo do processador que contém o processo em migração;
6) enviar mensagens pendentes: após a notificação de que o processo está estabelecido na nova locação, o núcleo fonte reenvia todas as mensagens que estavam na fila quando iniciou a migração, ou que chegaram desde o início da migração;
7) limpar o estado do processo: todo o estado do processo é removido e o espaço da memória e tabelas fica disponível. É deixado no processador fonte o endereço da nova locação do processo, para que o núcleo possa encaminhar corretamente as mensagens que porventura estavam ainda em trânsito, quando o processo foi transferido para um outro processador ( nesse instante o enlace é atualizado ). Feito isso, o núcleo fonte completa seu trabalho e o controle é retornado para o núcleo destino;
8) reiniciar o processo: o processo é reiniciado a partir do estado em que estava antes de ser migrado. As mensagens podem agora chegar para o processo, embora a única parte do sistema que conhece a nova locação está no núcleo fonte. Feito isso, o núcleo destino completa seu trabalho.

Os autores não abordaram os problemas de quando e para onde migrar um processo, se limitaram a apenas citá-los, como também não se preocuparam em analisar a migração de processo decorrente do colapso de um processador, se restringindo em apenas propor a infra-estrutura necessária para tal; no caso de uma memória estável.

## PROJETO CNET

O projeto Cnet iniciado em 1980 teve como diretivas o projeto e a implementação de um Sistema Distribuído baseado em uma arquitetura de rede local e para prover ferramentas e metodologias para a programação de aplicações distribuídas. O tipo de Sistema Distribuído considerado foi definido como uma federação de nodos ( hospedeiros )

autônomos frouxamente conectados se comportando como uma única entidade coerente com respeito à aplicação distribuída.

Nesse projeto, TIZATO e ZICARI ( 1985 ), se preocuparam em estudar mecanismos para a configuração dinâmica de Sistemas Distribuídos Localmente. Na realidade seus objetivos foram a de construir mecanismos que permitissem que aplicações distribuídas de usuários pudessem ser reconfigurados sem a necessidade de se parar toda a aplicação e que ele pudesse ser utilizado para balanceamento de carga, como também, para construção de sistemas tolerantes a falhas.

Apesar da proposta conter algumas restrições, mostra alguns caminhos bastante interessantes. Primeiro, somente as aplicações de usuários podem ser migrados, através de comandos emitidos pelo usuário programador; durante a execução da aplicação, o programador pode emitir por meio de seu terminal pedido para migração de alguns dos componentes computacionais da aplicação de um hospedeiro para um outro e, assim, reconfigurando-a dinamicamente. Segundo, todo componente computacional migrável, o seu código não é executado diretamente pelo "hardware" do processador e sim, interpretado, o que deverá limitar os tipos de aplicações, principalmente para aqueles em que o tempo é um fator importante e bastante restritivo. É claro que se deve levar em consideração a possibilidade da elaboração de técnicas para a construção de interpretadores e um conjunto de instruções primitivas, cujo tempo de execução de um programa escrito com elas possa ser comparável ao tempo que um mesmo programa necessitaria para ser executado diretamente pelo "hardware" do processador; a diferença de tempo não afete os propósitos da aplicação, estando ela escrito na linguagem nativa do "hardware" do processador, ou na de um interpretador. Dessa forma, apesar de ter sido esquematizado para aplicações de usuários, em primeira instância parece viável a aplicação dos "mesmos" conceitos e técnicas para a construção de Sistemas Operacionais.

Para isso, os autores utilizaram o conceito de máquinas abstratas, que eles denominaram de Nodos Virtuais. Cada Nodo Virtual possui sua memória, no caso, estruturada em quadros, e que serve para reter códigos ou dados; assim não existe o compartilhamento de memória entre eles. Suporta a execução de um conjunto de computações seqüenciais concorrentes, denominadas de atividades. Possui um conjunto de canais para a execução de operações de entrada e de saída e para a comunicação entre os nodos. Pode reagir a sinais síncronos ou

assíncronos, desde que ele esteja habilitado; isto é, no estado de pronto para execução. Por último, para todos os objetos ( atividades, quadros, etc ), de cada Nodo Virtual existe um descritor.

Uma aplicação pode consistir de vários desse Nodos Virtuais, distribuídos nos hospedeiros do Sistema Distribuído, sob as seguintes regras:

- um único Nodo Virtual não pode ser espalhado entre hospedeiros diferentes;
- mais do que um Nodo Virtual pode residir em um mesmo hospedeiro;
- os Nodos Virtuais não compartilham memória, mesmo que residam em um mesmo hospedeiro;
- a comunicação entre os Nodos virtuais pode ser feita apenas por meio de passagem de mensagens.

A comunicação entre Nodos Virtuais é feita por meio de portos locais unidirecionais, onde os nomes dos portos de entrada são visíveis ao meio externo. Posteriormente, segundo SEGRE ( 1987 ), foram incluídos outros tipos de portos e formas de comunicação, como o porto de difusão, a comunicação de forma não determinística, etc.

O ambiente suporte para execução de aplicações de usuários é oferecido por um Gerente de Recursos, os interpretadores e pelo Nodo Lógico. Todos eles residentes em cada hospedeiro de forma replicada. O Nodo Lógico, tem a função de dar o suporte a nível de subsistema de comunicação, como as primitivas de acesso à rede e a comunicação entre os hospedeiros, e outras que ficam melhor definidas nesse nível de hierarquia da estruturação do ambiente suporte. O gerente de recursos oferece primitivas para a criação, destruição, carregamento e descarregamento de Nodos Virtuais entre a memória e o meio de armazenamento secundário e para atualização e movimento de imagem de Nodos Virtuais para a migração. Tais pedidos devem ser feitas por meio de comandos emitidos pelo programador usuário. Oferece também primitivas para escalação de Nodos Virtuais e para gerenciamento de recursos, como a memória do hospedeiro, ligação dos recursos dos nodos e os portos. O interpretador é o processador dos Nodos Virtuais.

Nota-se assim que, essa esquematização, resolve pelo menos dois problemas envolvidos em uma migração de processo. Um seria a manutenção do mesmo ambiente de antes da migração ( pilha de trabalho, variáveis globais e locais, etc ), um outro seria quanto ao ajuste dos endereços das instruções componentes do código do

processo; sendo o código do processo interpretado não há tal necessidade.

## MOS: "A MULTICOMPUTER DISTRIBUTED OPERATING SYSTEM"

O Sistema Operacional MOS ( BARAK e LITMAN, 1985 ), é um sistema simétrico voltado para aplicações de uso geral em tempo compartilhado, projetado para um conjunto de computadores homogêneos e independentes, frouxamente conectados por meio de uma rede local. Esse projeto tem muitas semelhanças com o LOCUS ( POPEK et alii, 1981 ) em termos de objetivos, entretanto, difere em muito na estruturação. O sistema é caracterizado pela:

- multiplicidade de recursos;
- transparência de rede: a rede é completamente invisível;
- controle descentralizado: cada computador controla apenas seus recursos locais;
- autonomia: cada computador é operacional como um computador independente;
- disponibilidade;
- cooperação: os computadores cooperam entre si para prover serviços além dos limites da máquina e
- configuração dinâmica: computadores podem ser desconectados ou conectados à rede.

Na realidade o projeto consistiu, em primeira instância, por questões de simplificação de seu desenvolvimento, em substituir integralmente o núcleo do sistema UNIX versão 7 de cada um dos computadores, por um núcleo que oferecesse os seguintes objetivos funcionais: transparência da rede; disponibilidade e migração dinâmica de processos. Dessa forma, o MOS se comporta como um único sistema UNIX com o seu núcleo suportando todas as chamadas de sistema.

BARAK e LITMAN ( 1977 ) não se preocuparam com as questões de tolerância a falhas, diante do tipo de serviços que o sistema oferece; serviços gerais em tempo compartilhado. Se limitaram a considerar apenas o colapso de máquina, permitindo a sua isolação simples, perdendo os recursos ali existentes e, a migração de processos para poder reconfigurar dinamicamente o sistema para melhor balancear a carga. Para isso, o núcleo possui mecanismo para coletar lixo que deverá limpar do sistema todos os ponteiros perdidos que apontam para

a máquina que foi isolada; primitivas para permitir que o gerente do sistema desconecte uma máquina, retirando logicamente todos os processos ali residentes ( desconectando-os devidamente ) e; infra-estrutura para migração de processos que inclui mecanismo para transferência de grande quantidade de dados.

O núcleo do MOS foi estruturado em duas camadas. Uma denominada de núcleo inferior, o de mais baixo nível, e a outra de núcleo superior, que faz interface com os programas de usuários. O núcleo inferior é análogo ao de um sistema de multiprogramação, implementando os objetos locais e se responsabilizando pelas suas integridades. Além disso, oferece operações para o núcleo superior poder alterar o espaço de endereço virtual e sincronizar com eventos assíncronos e habilidade para que um elemento de enlace ( "linker" ), que reside no mesmo nível que este, possa executar operações sobre esse objetos locais.

O "linker" oferece ao núcleo superior suporte para que este faça chamada de procedimentos do núcleo inferior de qualquer computador do sistema. Embora o núcleo superior saiba onde reside o objeto destino, ele não necessita distinguir uma operação local de uma remota. O núcleo superior quando deseja um serviço do núcleo inferior, ele chama o "linker", passando para ele o identificador da máquina que contém o serviço solicitado e outros parâmetros necessários para a operação. É função do "linker" transmitir a solicitação do serviço para o núcleo inferior da máquina identificada. Note, então, que em MOS o "linker" serve como mensageiro, sendo responsável pelas transmissão dos pedidos de operações feitas pelo núcleo superior, porém é independente da semântica das operações. Ele é sensitivo à identificação da máquina para poder distinguir operações locais de remotos, tem conhecimento da configuração da rede e das máquinas que estão operacionais e implementa todo o protocolo de comunicação entre máquinas.

O núcleo superior implementa a interface entre o núcleo inferior e os processos que executam programas de usuários, escondendo destes todos os detalhes de "hardware", filtrando todas as chamadas de sistema.

## V.3 LINGUAGENS PARA PROGRAMAÇÃO DE SISTEMAS DISTRIBUÍDOS

Este item objetiva observar a importância das linguagens de programação de sistemas no contexto do presente trabalho e tecer opiniões a respeito de algumas linguagens existentes e disponíveis e, sugerir algumas características desejadas para que elas possam se constituir em uma ferramenta mais adequada para a implementação de "software" de Sistemas Distribuídos, no caso de "Sistemas Operacionais Distribuídos com Reconfiguração Dinâmica de Processos". Salienta-se que, não se pretende com isso, propor de forma categórica, características ideais de uma linguagem para programação de sistemas daquele tipo, e sim, apenas observar, pois se sabe que às vezes, determinadas características desejadas não são possíveis de serem implementadas eficientemente sem o auxílio de "hardware" muito especial, que no momento pode estar tecnologicamente indisponível ou, cujo custo é muito alto.

KIRNER ( 1986 ), faz uma análise com uma certa profundidade com respeito às características e requisitos de linguagens de programação de sistemas, em muitos aspectos como aqui desejadas. Em sua análise foram consideradas várias linguagens existentes e propostas, dentre as quais as seguintes: Pascal Concorrente ( HANSEN, 1977 ), MODULA ( WIRTH, 1977 ), Edison ( HANSEN, 1981 ), Mesa ( MITCHELL et alii, 1979; LAMPSON e REDELL, 1980; SWEET, 1985 ), DP ( HANSEN, 1978 ), ADA ( CHERRY, 1984; GEHANI, 1984; etc. ), CSP ( HOARE, 1978 ), PLITS ( FELDMAN, 1979 ), MODULA-2 ( WIRTH, 1983 ), SR ( ANDREWS, 1981, 1982 ), StarMod ( COOK, 1980; LEBLANC, 1984 ), Concurrent Euclid ( HOLT, 1983 ). SEGRE ( 1987 ), por sua vez se preocupou em analisar as linguagens sob os mesmos pontos de vistas que de KIRNER ( 1986 ), porém com maior atenção voltada para os aspectos de implementação. No caso, complementação da linguagem MODULA-2 com chamada remota de procedimento e da definição e implementação de uma linguagem de configuração para ambiente de programação distribuída baseada nesta linguagem. Outras linguagens, como a Pascal-m e a "Distributed Path Pascal" podem ser vistas em PAKER ( 1983 ) e FILMAN e FRIEDMAN ( 1984 ).

Dessa forma, a parte da análise das características das linguagens existentes e propostas para programação de sistemas, não serão aqui repetidas, principalmente, devidas essas referências serem

bastantes recentes. Assim, se fará apenas um resumo dos pontos mais relevantes segundo uma ótica um pouco diferente daquelas e em seguida complementadas, uma vez que se tem como maior preocupação aspectos de tolerância a falhas, de acordo com os capítulos anteriores.

Como é de conhecimento, a linguagem de programação é a ferramenta para a implementação de projetos de "softwares" para serem executados por sistemas de computadores. De forma análoga, um "software" é um produto manufaturado como outro qualquer, sendo necessário que ele seja feito para atender os requisitos do consumidor alvo. Para tanto é preciso que antes se faça uma pesquisa de mercado para colher informações a respeito desses requisitos, para depois então, especificar o produto, produzir e fazer os testes para verificar se ele atende plenamente às especificações. Satisfeito, ele pode ser colocado à venda, com o devido certificado de garantia e manutenção. Esse esquema é denominado de processo de desenvolvimento. Como o produto de interesse aqui é o "software", tal esquema recebeu o nome de processo de desenvolvimento de "software". Segundo GHEZZI e JAZAYERI ( 1982 ), esse processo é dividido em cinco fases, que se perceberá que segue o mesmo esquema descrito acima. Cada fase envolve uma atividade distinta e resulta em um conjunto de produtos claramente identificáveis. Essas fases são, geralmente, seguidas como cinco passos no desenvolvimento de "software". Entretanto, cada passo pode identificar deficiências nos anteriores, os quais devem ser repetidos. Essas fases são:

1- Análise de requisitos e especificação;
2- Especificação e projeto do "software";
3- Implementação ( codificação );
4- Certificação;
5- Manutenção.

De acordo com esse processo, a linguagem é necessária na fase 3. Como explicitado, as cinco fases são passos que devem ser seguidos de forma seqüencial. Assim, é desejável que as passagens entre elas sejam as mais suaves possíveis. Quer se dizer com isso que, como a especificação é descrita segundo uma metodologia de projeto, é natural que ela exerça uma grande influência na escolha da linguagem para a implementação. Em seguida, a implementação passa pela certificação que consiste de sua verificação contra a especificação. Feito isso, o "software" está pronto para ser utilizado. A manutenção é a fase de alteração do "software" devida a detecção de algum erro, ou mal

funcionamento de algum componente, ou devida à adição de novas capacidades ou para melhorar algumas antigas. Diante disso, nota-se bem o significado de seqüencial. Caso se esteja em uma fase e por algum motivo é preciso fazer alguma alteração em uma das fases anteriores, é preciso "abandonar tudo" que foi feito desde a fase que deve sofrer as mudanças, e recomeçar a partir desse ponto; pois, caso se começar a fazer remendos, provavelmente tornará as fases seguintes difíceis de serem elaboradas e, conseqüentemente, sujeitos a se cometer erros que com certeza frustrará o usuário. Colocou-se abandonar tudo entre aspas, porque isso depende do tipo de alteração. Pode ocorrer que a mudança é localizada com efeitos localizados também.

Há um bom tempo que se tem notado a influência das metodologias de especificação de projetos sobre as linguagens de programação. As linguagens consideradas modernas, incorporam muitos dos conceitos especificados por esse tópico da Engenharia de Software. Por exemplo, o mais antigo, construções de instruções para o computador descritas por meio de mnemônicos em vez na própria linguagem de máquina, exigindo com isso que elas sejam interpretadas ou compiladas para torná-las possíveis de serem entendidas pela máquina objeto. Quanto mais os mnemônicos são claros aos seres humanos, mais se afastam da compreensão da máquina real e assim, tais linguagens são ditas de alto nível. Outros conceitos como programação estruturada, esconder informações, abstração de dados e programação modular, foram sendo incorporados às linguagens de alto nível, sempre visando facilitar a tarefa da implementação do projeto e subseqüentes passos do processo de desenvolvimento de "software".

Apesar de ainda uma boa gama de tipos de aplicações, as linguagens como as Pascal, C, Mesa, FORTRAN, etc., todas em suas versões seqüênciais, são adequadas e eficientes, para muitas outras são deficientes, sendo necessário lançar-se mão de muitas construções de procedimentos descritos em linguagem de baixo nível, ou seja da "assembly" ( Linguagem Montadora ). Exemplos desse tipo de aplicação estão: os Sistemas Operacionais para Aplicações Gerais, controle de um sistema bancário, controle de processos, controle de caixas de supermercado com controle direto de estoque, etc.; enfim sistemas considerados de grande porte, que procuram compartilhar os limitados recursos de "hardware" do sistema de computação, de forma a obter deles o máximo de desempenho e, além disso, serem confiáveis. Cabe observar que, as linguagens C e Mesa, já foram elaboradas contendo

muitas das facilidades descritas mais adiante, como também as versões mais recentes de Pascal, pois seus objetivos iniciais eram para servir como linguagens para programação de sistemas desses tipos.

Para esses últimos tipos de aplicações, foram construídas algumas linguagens de alto nível que, além de incorporarem os conceitos já consagrados, como programação estruturada e abstração de dados, incorporam outros mais recentes como, programação modular e compilação em separado. Além disso, a fim de evitar o uso da "assembly", algumas delas oferecem facilidades de acesso a recursos de baixo nível, como definição de endereço absoluto, acesso para leitura e escrita dos registradores da máquina, etc., como também oferecem construções lingüísticas para implementação de exclusão mútua, implementação do não determinismo para recepção de mensagens, rotinas para comunicação e sincronização entre processos baseada ou não em troca de mensagens, etc. Enfim, de modo global, procurando que o sistema possa ser implementado por meio de uma única linguagem de alto nível.

Tais facilidades ou são suportadas pelo núcleo da linguagem ( suporte em tempo de execução - "Run Time Support" ) e/ou são suportadas por bibliotecas. As facilidades que podem ser suportadas por meio de biblioteca são aquelas que são utilizadas como tipos, constantes e rotinas; enfim aquelas que não exigem construções estruturadas para suas utilizações diretas. Já as que exigem uma construção estruturada como as para implementação de exclusão mútua, para implementação do não determinismo, devem ser suportadas pelo núcleo da linguagem; ou seja devem fazer parte da sintaxe da linguagem. Como exemplo de linguagem mais difundida que incorpora as características e facilidades descritas é a linguagem ADA ( CHERRY, 1984; GEHANI, 1984 ), a qual, oferece várias outras além dessas, como por exemplo, facilidade para tratamento de exceções.

Uma linguagem que foi criada na mesma época que a ADA, foi MODULA-2 devida a WIRTH ( 1983 ), a qual em vez de oferecer todos os recursos como aquela, se preocupou em apresentar os suficientes para que, a partir deles o usuário possa construir seus próprios mecanismos e respectivas construções.

Como destacado, a escolha da linguagem para ser utilizada para a implementação, depende da especificação do projeto, pois, às vezes, a utilização de uma linguagem mais flexível ( por exemplo, MODULA-2 ) pode ser mais adequada, apesar de exigir maior esforço de

programação. Entretanto, como ADA, essa apresenta a facilidade da "re-usuabilidade de software" ( utilização direta do "software" do jeito que está, sem a necessidade de se fazer qualquer adaptação ), por meio da compilação em separado, o que implica que o esforço é exigido uma única vez. Uma vantagem dessa linguagem em relação à ADA é a sua simplicidade e conseqüente porte.

Até agora, discutiram-se os conceitos, facilidades de baixo nível e mecanismos suporte para comunicação e sincronização, que algumas linguagens de programação de sistemas oferecem. Entretanto, em termos de confiabilidade, elas deixam a desejar, pensando em utilizá-la sem a necessidade de se criar novos recursos; pela própria linguagem ou por meio da "assembly". Assim, listam-se alguns recursos ( veja capítulo IV ) que se julga interessantes que as linguagens ofereçam para a construção de sistemas tolerantes a falhas, como também, os recursos de "hardware" que porventura seja necessários para dar o devido suporte:

- tratamento de exceções: construções para levantamento e tratamento de exceções ( Ada apresenta );
- memória estável: memória não volátil de leitura e de escrita de velocidade comparável a de memória principal, para o armazenamento de informações que não deverão ser destruídas para permitir a recuperação do sistema;
- construção de blocos de recuperação: como a sugerida por RANDELL ( 1975 );
- construções de ações atômicas: por exemplo, como descrita por LOMET ( 1977 );
- construção para o estabelecimento de pontos de verificação: para a construção de linhas de recuperação;
- facilidade para a execução simulada: para permitir a recuperação de processos cujas operações efetivadas não sejam fáceis de serem desfeitas, por exemplo, mensagens do tipo comando já enviadas, recebidas e executadas;
- etc.

Outros recursos, como por exemplo, mecanismos confiáveis de comunicação e sincronização entre processos baseados no conceito de portos, linguagem de configuração, etc., já mencionados e que serão novamente abordados nos capítulos seguintes.

# CAPÍTULO VI

## DESENVOLVIMENTO DE UM SUPORTE PARA CONSTRUÇÃO DE SISTEMAS OPERACIONAIS DISTRIBUíDOS TOLERANTES A FALHAS

Neste capítulo discutem-se e descrevem-se os mecanismos de tolerância a falhas que constituem na principal contribuição da tese e que serão utilizados no projeto de um suporte para Sistemas Operacionais Distribuídos com Reconfiguração Dinâmica de Processos, apresentado a seguir deste, para fins de exemplificação.

Dos vários tipos de falhas que podem ocorrer em um Sistema Distribuído, se está preocupado com as que ocorrem no subsistema de comunicação e nos nodos operadores, mais especificamente, colapsos destes. Não se tratarão falhas ocasionadas por defeitos de projeto ( falhas de "software" do tipo não antecipáveis, como descrito no item III.4.1 ), como também, falha em um nodo operador, decorrente do mau funcionamento ( eventual ou não ) de algum de seus componentes não vitais, permitindo o seu funcionamento e que, porém, afetam os processos ali residentes, fazendo com que eles se desviem de suas especificações ( veja capítulo III ).

Para tanto, existe a preocupação com o desenvolvimento dos seguintes mecanismos:

- mecanismos para recuperação de processos: para tratar colapso de nodo operador ou de sua inacessibilidade, decorrente de defeito no

nodo de comunicação ou de rompimento de linhas de comunicação que isolam tal nodo ( por exemplo em uma rede com conexão ponto a ponto - quase ou totalmente conectada );

- mecanismos para migração de processos: visto mais como suporte para balanceamento dinâmico de cargas, é necessário como suporte para recuperação de processos ( recuperação de um processo residente em um nodo para um outro ) e para tolerar falhas antecipáveis como, por exemplos: para o operador do sistema poder desativar determinado nodo operador para poder colocá-lo à disposição do pessoal de manutenção preventiva; desativação involuntária do nodo operador, mas que contenha uma fonte de energia ( "NO-BREAK" ) suficiente para executar a tarefa de migração dos processos ali residentes; etc.
- mecanismos para comunicação e sincronização de processos tolerantes a falhas: para tratar falhas que possam ocorrer durante uma comunicação entre processos, como, falhas transientes do meio físico de comunicação, colapso do processo destino ou do emissor, que, dependendo do tipo de comunicação ( síncrona ), pode dar surgimento a mensagens e computações órfãs, que deverão ser detectadas e removidas do sistema, pois no mínimo, elas desperdiçam recursos, podendo até interferir nas tarefas de processos corretos ( SHRIVASTAVA, 1983 - item IV.5 ).

Julgam-se que tais mecanismos compõem um suporte adequado para a construção de Sistemas Operacionais Distribuídos, com a característica de ser reconfigurável dinamicamente, no sentido de tolerar aqueles tipos de falhas.

Desses, a maior contribuição se concentra no desenvolvimento de um mecanismo para comunicação e sincronização entre processos baseado em portos com suporte para a detecção e eliminação de órfãos. Nos outros mecanismos as contribuições são, proporcionalmente, menores. O mecanismo para recuperação de processos será utilizado também como suporte para a eliminação de computações órfãs, e o para a migração de processos, além de o complementar, oferecerá as facilidades descritas no item IV.4.

Assim sendo, inicia-se por uma discussão sobre os mecanismos para recuperação de processos, passando pelos mecanismos para migração de processos, seguido da discussão sobre tratamento de órfãos, para fechar com a descrição da proposta do mecanismo de

comunicação e sincronização entre processos baseado em portos, com suporte para detecção e eliminação de órfãos.

## VI.1 MECANISMO PARA RECUPERAÇÃO DE PROCESSOS

Nos capítulos anteriores discutiram-se as técnicas para estruturação de Sistemas Operacionais Distribuídos Tolerantes a Falhas e mecanismos e estruturas de programas propostos para impor essa característica, objetivando que a construção de tal "software" não se torne muito mais complexo que para o desenvolvimento de um, sem essa característica desejada. Entretanto, muitas dessas propostas acabaram esbarrando na necessidade de dispositivos especiais. Por exemplo, o cache de recuperação exigido pelo Bloco de Recuperação, ou um dispositivo votador totalmente confiável no esquema da maioria votante. Com isso, muitas pesquisas teóricas e práticas foram e continuam sendo feitas, procurando soluções alternativas, tanto a nível de "hardware" como de "software". Disso, muitas delas apresentam simplificações, ou melhor, impõem restrições, postergando soluções para elas ou exigindo cuidados especiais, que, porém, não as invalidem. Por exemplo, no esquema da programação N-Versões, onde se confia na integridade do procedimento votador.

Aqui se está interessado em construir um mecanismo confiável para comunicação entre processos, que trate falhas de sub-sistema de comunicação e de nodos operadores e em um mecanismos que ofereça a facilidade para a migração de processos, para servir como suporte para o desenvolvimento de Sistemas Operacionais Distribuídos com Reconfiguração Dinâmica de Processos.

Para o primeiro mecanismo, como melhor se verá no item VI.3, falhas acima citadas podem dar origem a mensagens e computações órfãs, diante das formas propostas e utilizadas para os seus tratamentos. Isso não quer dizer que elas impõem o aparecimento desses problemas, como será visto neste item. Uma proposta encontrada na literatura que lida com esses tipos de falhas é aquela discutida no item IV.5 ( PANZIERI e SHRIVASTAVA, 1988 ); um mecanismo de chamada remota de procedimento, que foi denominado de Rajdoot. Nessa, entretanto, observou-se que ela simplesmente garante que não existirá nenhuma computação órfã ativa, quando um processo fizer uma nova tentativa de chamada ( eliminando-as primeiro antes de atender essa

chamada ) e que computações órfãs cujos resultados não mais são de interesse para o sistema, serão alguma hora eliminadas. Para isso, cada chamada cria um processo no nodo operador que mantém o procedimento chamado, que terá as funções de fazer a chamada de forma local e de enviar respostas ( no mínimo, para avisar o término de suas tarefas ) para o processo chamador. Dessa forma, o mecanismo proposto se limita a destruir esses tipos de processos, quando da necessidade de eliminar computações órfãs, deixando a nível de processo a responsabilidade de, ou não, desfazer os efeitos de suas operações. Uma proposta de chamada remota de procedimento que incorpora um mecanismo para esse fim é a devida a LIN e GANNON ( 1985 ), denominada de Chamada Remota Atômica de Procedimento, que segue a semântica "quanto muito uma vez". Entretanto, segundo Panzieri e Shrivastava, esse esquema é dispendioso e portanto pode não ter interesse em muitas aplicações.

Na presente proposta, entretanto, tais processos criados naquele mecanismo não existirão, simplesmente porque, agora, a comunicação e sincronização deve ser feita diretamente entre elementos ativos ( os processos ). Assim, mesmo admitindo uma comunicação síncrona como a chamada remota de procedimento, as técnicas de eliminação de computações órfãs diferirão bastante, principalmente, ao se levar em consideração as várias formas de comunicação que os processos poderão desejar estabelecer, segundo os modelos cliente x servidor e processos cooperantes, a serem discutidas no item VI.3. Antecipando, para eliminar a computação órfã que executou operações idempotentes, basta recolocar o processo no estado anterior ao início da execução dessa computação. Quanto às técnicas para desfazer os efeitos das operações ( por elas não serem idempotentes ), serão discutidas no item VI.3.

Assim sendo, nota-se a necessidade da existência de um mecanismo que dê o suporte para aquele para comunicação entre processos, para que este possa solicitar os serviços de recuperação de um processo no sentido de eliminar computação órfã, pelo menos, com os mesmos efeitos que os do Rajdoot. Para tanto, em vez de criar um mecanismo específico para esse fim, optou-se pela utilização do esquema proposto por POWELL e PRESOTTO ( 1983 ), discutida no item IV.3.2, apesar da premissa admitida de que os processos devem ser determinísticos, ou seja, a recuperação de um processo não exigirá que os efeitos de suas operações executadas entre o ponto de recuo até o instante da sua suspensão para esse fim, sejam desfeitos e,

conseqüentemente, não afetando outros processos. Os motivos por essa opção foram os seguintes, em ordem:

- não exigir dispositivos especiais não encontrados facilmente no mercado;
- pela simplicidade relativa;
- objetivar servir de suporte a mecanismos baseado em troca de mensagens de níveis mais alto; e
- se constituir em um mecanismo para recuperação de processos que entre em ação logo que ele próprio detecte alguma falha em um processo do sistema.

É bom lembrar, entretanto, que a proposta postergou explicitamente descrever e implementar o caso em que a recuperação de um processo deve ser feita em algum outro nodo operador. Além disso, implicitamente, admitiu as seguintes hipóteses, quando o mecanismo entrar em ação:

- as suas mensagens enviadas para fim de recuperação de um processo, não se perdem, como também, as mensagens respostas; e
- um processo em recuperação não entra em colapso.

Dessa forma, tal mecanismo deverá ser alterado para suprimir essas duas hipóteses, como também, deverá ser complementado para incorporar o suporte para recuperação de processos no caso de ocorrência de colapso de nodos operadores ( migração de processos ) e com outras funções que serão indicadas quando da discussão sobre o mecanismo para tratamento de órfãos e respectiva apresentação da proposta objeto da tese. Nesse sentido, a seguir, discute-se a incorporação de um mecanismo para migração de processos, visando a complementação desse para recuperação de processos, como também, um para atender às metas originais de tal tipo de mecanismo.

## VI.2 MIGRAÇÃO DE PROCESSOS

No item IV.4 descreveu-se a definição de migração de processos, a importância da colocação dessa facilidade em Sistemas Distribuídos, como mecanismos para permitir o balanceamento dinâmico de cargas, para possibilitar o crescimento incremental e para aumentar a sua confiabilidade ( tolerância a falhas ), como levantada na introdução deste capítulo e alguns dos problemas encontrados para a implementação de mecanismos para a execução de tal tarefa.

Relembrando, a migração de um processo consiste, basicamente, em mover o código do processo e seu estado de um nodo operador para um outro. Para isso, é necessário se preocupar com algumas questões, como: pode ocorrer que quando um processo está migrando, existam processos que enviaram mensagens para ele em seu endereço antigo ( mensagens em trânsito ); a temporização prevista para o envio síncrono, deve ser congelada, a fim de evitar que o seu estouro levante uma falsa exceção; etc. Preocupado com a questão referente ao problema das mensagens em trânsito, quando da execução da tarefa de migração, POWELL e MILLER ( 1983 ) - Migração de Processos em DEMOS/MP - propuseram um mecanismo descrito resumidamente naquele item citado.

No presente trabalho, objetiva-se propor um mecanismo para migração de processos para esses fins e um para aquele anotado no item anterior, ou seja, para complementar o mecanismo para recuperação de processos, devido a POWELL e PRESOTTO ( 1983 ).

Entretanto, observado que todos os dados necessários para a realização da migração de um processo já se encontram disponíveis naquele mecanismo para recuperação de processos, exceto a informação sobre qual o nodo destino, "basta" implementar as funções específicas para aqueles fins; naturalmente, suprindo-o, primeiro, daquela informação, por exemplo, a ser fornecido pelo balanceador de carga.

Para tanto, será utilizado como diretriz o esquema proposto para o DEMOS/MP. Observa-se que nesse não foram admitidas as hipóteses de ocorrência de falhas no sistema de comunicação, como também de colapso de nodos operadores destinos, cuidados destacados nos próximos itens deste capítulo.

Assim, para que o mecanismo para recuperação de processos possa executar uma migração com fins de recuperação, será necessário incorporar as seguintes funções e informações, em termos gerais ( os detalhes serão vistos mais adiante ):

- informação sobre em que nodo operador deverá ser recuperado o processo;
- transferir de seu banco de cópias de segurança o código do processo para o nodo operador destino; e
- rotear todas as mensagens destinadas ao processo em seu endereço antigo para o novo e, ao mesmo tempo, informar os processos fontes do novo endereço.

Já para que ele execute uma pura migração de processos, basta que se incorpore nele a seguinte função, além daquelas: atender pedido

para migração de processos, que poderia vir ou não explicitando o nodo destino, que implica como primeiro passo, o salvamento de ponto de verificação do processo a ser migrado, para evitar a necessidade da emissão das mensagens pelo mecanismo de recuperação para aquele processo; maiores esclarecimentos estão no item VI.4.1.

A seguir, entra-se na discussão sobre o tratamento de órfãos.

## VI.3 MECANISMOS PARA TRATAMENTO DE ÓRFÃOS

Órfãos são computações de um Sistema Distribuído que são escalonadas devido a ocorrência de vários eventos indesejáveis tais como duplicação de mensagens e colapsos de nodos operadores ou de comunicação. No item IV.5, descreveram-se as possibilidades do surgimento desses eventos indesejáveis, como também, as técnicas para suas detecções e remoções do sistema ( tratamento de órfãos ). Tais técnicas foram devidas a PANZIERI e SHRIVASTAVA ( 1988 ), para suas incorporações em um mecanismo de chamada remota de procedimento, que eles denominaram de Rajdoot e que foi projetado para Sistemas Distribuídos consistindo de "clientes" e "servidores".

Agora, estudam-se o surgimento desses problemas e a aplicabilidade dessas técnicas em mecanismo para comunicação e sincronização entre processos baseado em portos locais. Para tanto, a seguir, discutem-se essas técnicas, passo a passo, com vistas a esses objetivos, considerando que a comunicação entre processos seja síncrona com resposta associada, que é o caso de chamada remota de procedimentos. Para o caso de comunicação assíncrona, a discussão será feita quando da descrição daquele mecanismo de comunicação e sincronização entre processos; no próximo item.

Quanto ao tratamento de mensagens duplicadas e mensagens órfãs, é perceptível que as técnicas utilizadas podem ser as mesmas; enumeração seqüencial das mensagens.

Quanto às computações órfãs, no esquema de Rajdoot, elas são detectadas e abortadas, por meio de três mecanismos, conforme visto no item IV.5, a saber: o "deadline", o "crash-counter" e o processo exterminador. O primeiro para garantir que uma chamada terminada anormalmente, na ausência de colapso, não deixe computações órfãs. O segundo, para garantir que na ocorrência do colapso do nodo cliente e quando de sua recuperação fizer uma chamada decorrente desse fato,

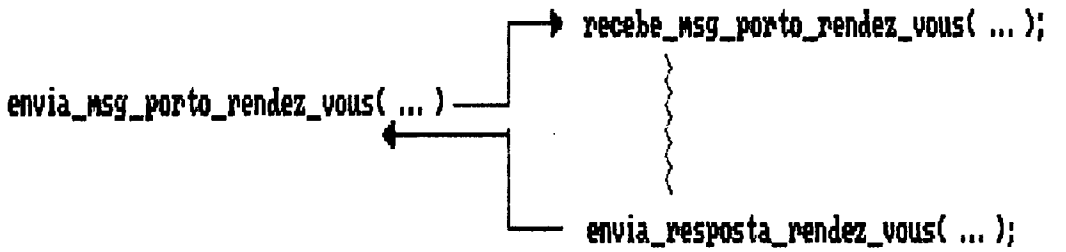
as computações órfãs da chamada anterior sejam devidamente abortadas, antes de executar esta. Por último, para o caso em que o nodo que entrou em colapso nunca mais ( ou muito tarde ) seja restaurado ou que não mais faça chamada de procedimentos residentes no nodo que antes fazia, é garantido que todas as computações órfãs serão uma hora abortadas.

Lembre-se que esses mecanismos para tratamento de órfãos foram projetados para o mecanismo de chamada remota de procedimentos, o qual é de nível mais alto que aquele que se pretende implementar; comunicação e sincronização baseado em portos. Dessa forma é necessário primeiro analisar as possibilidades de surgimento de computações órfãs quando se utiliza esse tipo de mecanismo de comunicação e sincronização entre processos.

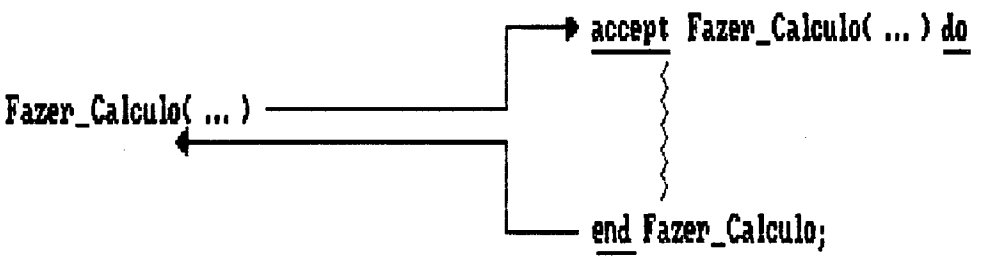
Para as comunicações síncronas, na forma de "rendez-vous" simples, não surgem computações órfãs, pois quando do estabelecimento da sincronização, toda a tarefa da comunicação já foi realizada. Entretanto, para o outro caso, quando a comunicação e sincronização é feita no esquema de "rendez-vous" estendido, há a possibilidade do surgimento de computações órfãs, por, justamente, a extensão se comportar como a execução de um procedimento. Por exemplo, pode-se fazer com que o "rendez-vous" se estenda pelo processo inteiro, transformando este, praticamente, em um procedimento. Assim, para esse tipo de comunicação e sincronização, aparentemente, podem-se usar os mesmos três mecanismos citados. Obviamente, eles poderão não ser idênticos, seus efeitos é que devem ser os mesmos, pois, no caso de chamada remota de procedimentos Rajdoot, à cada primeira chamada feita por um processo cliente para um servidor é criado um processo servidor particular para esse cliente. Dessa forma, cada cliente terá seu processo servidor. Logo, a eliminação de um processo criado com essa finalidade, representa o aborto da computação feita pelo serviço chamado; caso essa computação deixe resíduos, estes devem primeiro serem devidamente eliminados, cuja tarefa no Rajdoot é deixada para o "software" de nível superior ( por exemplo, para o processo cliente ). Com isso, detectada uma computação órfã pelos mecanismos de "crash-counter" e exterminador, basta eliminar os processos desse tipo. Já para aquele mecanismo de troca de mensagens entre processos, não é criado nenhum processo para a finalidade do estabelecimento da comunicação, assim, tais processos órfãos não surgirão. Além disso, terminado de executar o trecho síncrono da comunicação ( a extensão

do "rendez-vous" ), a execução do processo continua. Diante disso e de que existe o suporte para recuperação de processos, analisam-se a seguir os possíveis problemas que o estouro de "time-out" de chamada e os colapsos de processos clientes e servidores ( ou processos cooperantes ), podem ocasionar, por exemplo, o aparecimento de computações órfãs. Para tanto, antecipa-se que a comunicação e sincronização na forma de "rendez-vous" estendido, por meio de mecanismos de troca de mensagens baseado em portos, será feita por meio do uso de três primitivas, a saber:

- **envia_msg_porto_rendez_vous ( porto, msg, tempo, status, ... )**: envia mensagem para seu porto local e se suspende à espera de resposta. Um mecanismo subjacente se encarrega de transmitir a mensagem ao processo destinatário, colocando-a em seu porto e verificar se o processo está suspenso à espera de mensagem. Se estiver, ativa-o, caso contrário, não faz mais nada;




( a ) Modula 2




( b ) Ada

Figura VI.1: Esquemas de comunicação e sincronização entre processos na forma de "rendez-vous" estendido.

- **recebe_msg_porto_rendez_vous ( porto, msg, ... )**: verifica se existe mensagem em seu porto local, se existe, recebe-a e executa as operações que seguem. Caso contrário, se suspende;
- **envia_resposta_rendez_vous ( porto, resposta, ... )**: envia mensagem resposta à mensagem recebida pelo seu par, o recebe_msg_porto_rendez_vous, para o processo emissor e ativa-o;

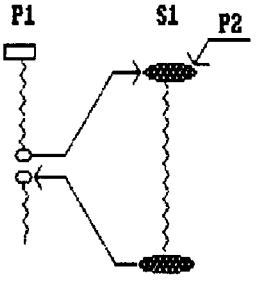
Note assim que, se uma linguagem de programação não as incorporarem como construções próprias e as utilizá-las dessa forma, o programa não deixará tão claro que determinado trecho corresponde à extensão do "rendez-vous". Para maior clareza, a figura VI.1 (a) ilustra esse caso e a figura VI.1 (b), o outro.

### VI.3.1 ANÁLISE DOS PROBLEMAS

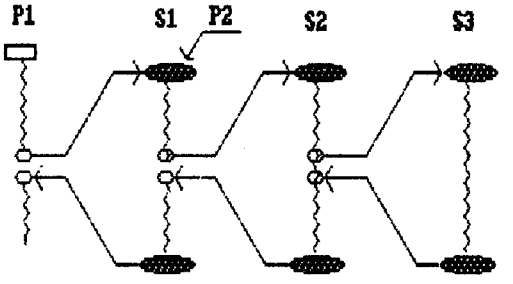
Analisam-se a seguir os possíveis problemas de comunicação na forma de "rendez-vous" estendido, que o estouro de tempo limite e os colapsos de processos emissores ( ou clientes ) e receptores ( ou servidores ), podem ocasionar o aparecimento de computações órfãs. Para tanto, primeiro ilustram-se os possíveis modos de cooperação entre processos, se na forma de cliente x servidor ou na forma de processos cooperantes. Em seguida, descreve-se o modelo de falhas, para depois fazer a análise propriamente dita, sempre iniciando pela forma de cooperação mais simples.

## A) FORMAS DE COOPERAÇÃO

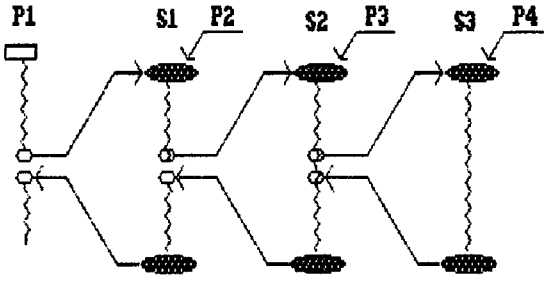
### A.1) MODELO PROCESSO CLIENTE X PROCESSO SERVIDOR




(a) Vários clientes usando um único servidor




(b) Vários clientes usando vários servidores de forma puramente aninhada

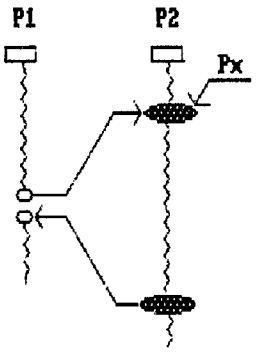



(c) Vários clientes usando vários servidores de forma aninhada, porém cada um podendo ser compartilhado por outros processos clientes

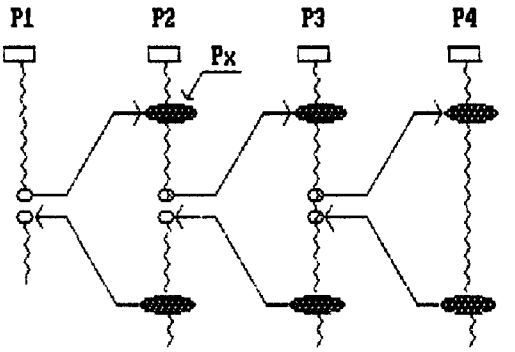
Figura VI.2: Formas de Cooperação entre Processos Clientes e Processos Servidores: P - simboliza processo cliente S - simboliza processo servidor.

$P_1$ precisa dos serviços de $S_1$, $S_2$ e $S_3$
$P_2$ precisa dos serviços de $S_1$, $S_2$ e $S_3$
$P_3$ precisa dos serviços de $S_2$ e $S_3$
$P_4$ precisa dos serviços de $S_3$

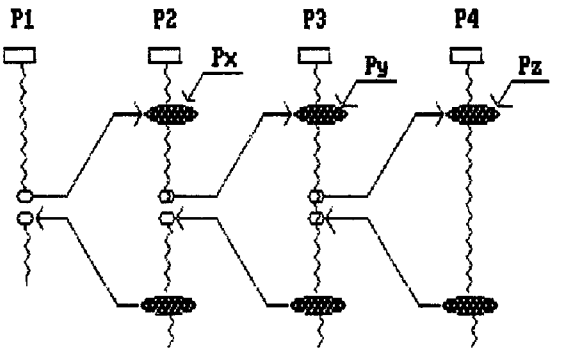
## A.2) MODELO PROCESSOS COOPERANTES




(a) Vários processos se comunicando com um único




(b) Vários processos se comunicando de forma puramente aninhada




(c) Vários processos se comunicando de forma aninhada

Figura VI.3: Formas de Cooperação entre Processos por meio do esquema de "rendez-vous" estendido: P - simboliza processo

## B) MODELO DE FALHAS QUE PODEM DAR SURGIMENTO A COMPUTAÇÕES ÓRFÃS

Uma comunicação pode falhar significando o término anormal de um envio de mensagem, pelos seguintes motivos:

- falha de comunicação: não existe o destino, por defeito no nodo de comunicação ou do operador que o contém;
- o nodo operador onde reside o processo emissor da mensagem apresenta um defeito logo após a sua emissão; e

- o nodo operador onde reside o processo receptor da mensagem apresenta um defeito logo após recebê-la;

É admitida a hipótese de que a memória de um nodo operador ou de comunicação nunca falhe. Apesar de que, no caso da memória de nodo operador, o gerente de memória poderia detectar a invasão destrutiva de um espaço de memória. Por exemplo, um processo querer alterar o conteúdo de uma posição de memória pertencente a um outro processo sem a devida autorização, ou seja sem ser por meio de mecanismos específicos para isso ( mecanismo de comunicação, por exemplo ). Isso pode acontecer ( admitindo que os processos foram construídos sem nenhum defeito de projeto ), caso apareça um defeito na memória que provoque a alteração indevida de uma ou mais posições de memória, que por sua vez, leva ao fato acima citado. Como também, pode ser que essa invasão indevida não aconteça. De qualquer forma, ao salvar o estado de um processo pode-se incorrer no erro de salvar o estado de um processo já deteriorado. Um meio seguro para recuperar um processo que falhou devido a esse tipo de defeito, é colocá-lo em seu estado inicial. Entretanto, como visto, essa técnica de recuperação é muito cara, em termos de tempo de processamento. Além disso, provavelmente, esse erro, contaminou outros processos; salvo se cada processo se constitua em um único programa ( multiprogramação ). Uma outra forma para detectar um defeito de memória poderia ser por meio da utilização de blocos de recuperação ou uma espécie de. Por exemplo, cada processo estaria encapsulado por uma espécie de bloco de recuperação. Se a computação feita passasse pelo teste de aceitação, então, ela estaria em ordem. Caso contrário, o processo se perdeu. Admitido que não existe erro de programação, a falha é proveniente de alguma coisa que não funcionou direito. Assim, a tal espécie de bloco de recuperação poderia solicitar a recuperação de si próprio, naturalmente, em uma outra área da memória e solicitar ao gerente de memória que inutilize aquela. Nota-se que, tal recuperação, não precisa ser da forma colocá-lo em seu estado inicial, pois aquela espécie de bloco de recuperação, após passar pelo seu teste de aceitação, pode pedir o salvamente do estado, garantindo que este é de um processo que estava em operação normal, dentro de suas especificações. Essa forma, porém, deve considerar que o teste de aceitação nunca é deteriorado.

## C) DETECÇÃO E ELIMINAÇÃO DE COMPUTAÇÕES ÓRFÃS

Diante da existência, de mecanismo de recuperação de processos, discutido neste capítulo, procura-se a seguir analisar as técnicas para detecção e eliminação de computações órfãs, descritas no item IV.5, para um mecanismo de comunicação e sincronização entre processos, baseados em portos locais. Mais especificamente, quando a comunicação e sincronização é feita na forma de "rendez-vous" estendido.

Para tanto, inicia-se tal objetivo, à luz das formas de cooperação entre processos, em ordem, ilustradas nas figuras VI.2 e VI.3.

### C.1) TÉRMINO ANORMAL DE ENVIO DE MENSAGEM DEVIDO AO ESTOURO DO TEMPO LIMITE ( SEM A OCORRÊNCIA DE COLAPSO DE NENHUM DOS PROCESSOS ENVOLVIDOS )

Como sabido, a utilização de um tempo limite para o término de um envio síncrono de mensagem é para evitar que o processo emissor fique suspenso para sempre, no caso de não receber nenhum sinal ou resposta que confirme que tal mensagem foi recebida.

Pela figura VI.4, os eventos que podem ter ocasionado que o processo $P_1$, tenha sido ativado por estouro do tempo limite são:

(a) o tempo limite estabelecido pelo emissor foi insuficiente;

(b) o envio da resposta pelo servidor $S_1$, não chegou ao $P_1$;

O evento (b), por hipótese, somente pode ocorrer se houve colapso do nodo operador de $S_1$; caso a ser analisado posteriormente.

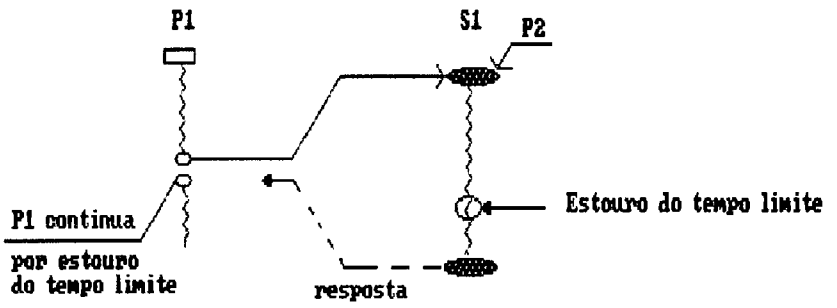



Figura VI.4: Término anormal de comunicação por estouro do tempo limite da emissão de uma mensagem para um processo servidor

Assim, as alternativas que $P_1$ pode seguir são:

(1) $P_1$ pode querer fazer uma nova tentativa com um tempo limite maior;

(2) $P_1$ pode querer emitir a mesma mensagem para um processo servidor alternativo ou fazer qualquer outra coisa;

(3) $P_1$ pode levantar uma exceção, solicitando a recuperação do processo servidor, imaginando que ele entrou em colapso;

Diante disso, nota-se que podem surgir os seguintes problemas:

(i) Se realmente o tempo limite foi insuficiente: a nova tentativa deixa a computação anterior órfã;

(ii) Se a opção de $P_1$ for a (2): a computação anterior fica órfã;

(iii) Se a opção for a (3): considerando o esquema de recuperação adotado ( POWELL e PRESOTTO, 1983 ), o qual evita a emissão e a recepção de mensagens já devidamente efetivadas, pode acontecer de $P_1$ entrar em execução cíclica sem fim, se o tempo limite não for mais levado em consideração.

## C.1.1) DETECÇÃO E ELIMINAÇÃO

A técnica empregada por Rajdoot foi em utilizar um valor calculado a partir do tempo limite definido na primitiva de envio de mensagem, que determinará o tempo máximo que o processo servidor tem, para enviar uma resposta e esta chegar ao processo emissor, o qual foi denominado de "deadline" ( tempo limite para execução ). Tal técnica foi empregada, justamente, para detectar e eliminar computações órfãs; não todas, entretanto, como se verá no decorrer da discussão.

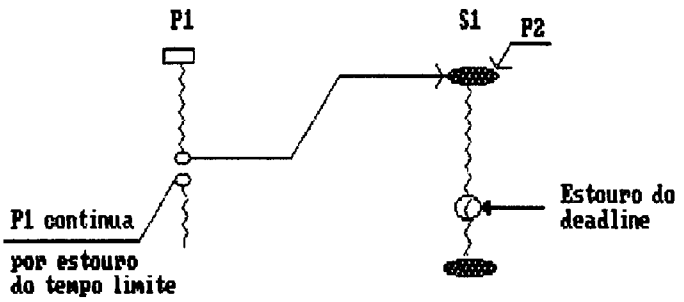



Figura VI.5: Término anormal de comunicação por estouro do "deadline" => estouro do tempo limite da emissão de uma mensagem para um processo servidor

A idéia básica dessa técnica é de que, quando expirar o tempo estabelecido pelo "deadline", expirou o tempo limite do emissor da mensagem. Dessa forma, se está ciente de que o processo emissor foi ativado pelo mecanismo do tempo limite e, a princípio, tem a liberdade de fazer qualquer coisa, como aquelas descritas em C.1. Assim, o próprio mecanismo de comunicação e sincronização, por intermédio da primitiva envia_resposta_rendez_vous, poderá tomar as devidas providências com respeito à computação realizada pela extensão do "rendez-vous".

## ALTERNATIVAS DE SOLUÇÃO

(1) A envia_resposta_rendez_vous simplesmente não envia nenhuma resposta e levanta tal exceção para que o processo que a executou tome as devidas providências. Por exemplo:

(a)
```
repita
    recebe_msg_rendez_vous ( ... );
            .
            .
    envia_resposta_rendez_vous ( ..., T );
        (* T = Verdade se estourou o "deadline";
           T = Falso, caso contrário *)
até que T;
```

(b)
```
recebe_msg_rendez_vous ( ... );
        .
        .
envia_resposta_rendez_vous ( ..., T );
se T é falso então
        executar medidas saneadoras;
fim_se;
```

Note que a solução (a) fica ignorando as computações realizadas, até que uma mensagem recebida ofereça um "deadline" suficiente para que o envia_resposta_rendez_vous consiga enviar a mensagem resposta ao processo emissor. Dessa forma, percebe-se que ela é boa para os casos ilustrados nas figuras VI.2 e figuras VI.3 (a) e, (b) e (c) se o estouro do "deadline" for percebido em $P_4$, desde que as computações

sejam idempotentes. Caso as computações não sejam idempotentes, seus efeitos permanecerão e, assim, refletirão nas posteriores. Preocupação mostrada pelo exemplo (b), o qual poderia estar encapsulado por um comando de repita como em (a). Se, porventura, o programador resolver, ele próprio, programar as operações de desfazer as computações realizadas, ele deverá se preocupar em ver como o mecanismo de recuperação subjacente funciona. Isso será melhor notado logo adiante.

(2) O mecanismo de comunicação e sincronização incorpora algumas medidas para tolerar esse tipo de falha. A recebe_msg_rendez_vous ao receber uma mensagem, marca um ponto de recuperação ( ou verificação ) particular que a envia_resposta_rendez_vous possa acessar. Dessa forma, essa primitiva ao detectar que o "deadline" expirou, terá condições de abortar as operações executadas, o que é feito ao colocar o processo receptor no estado daquele ponto de verificação. Em seguida ela pode destruir tal ponto de verificação e dar continuidade ao processamento, colocando o processo pronto para receber uma nova mensagem. Cabe observar que, como o mecanismo de recuperação de processos guardou na cópia do processo receptor a mensagem que foi descartada pela recuperação feita pela primitiva envia_resposta_rendez_vous, como também incrementou o contador de mensagens já enviadas e recebidas na cópia do respectivo processo emissor, a envia_resposta_rendez_vous terá que emitir uma mensagem para o gerente de recuperação solicitando que ele atualize essas informações das correspondentes cópias; isto deve ser feito para manter a integração do mecanismo de comunicação e sincronização com o mecanismo de recuperação.

Nota-se que tal esquema funciona para os casos de comunicação ilustrados pelas figuras VI.2 (a) e VI.3 (a). Já para atender também o caso da figura VI.2 (b), o esquema fica mais complexo, pois necessitará salvar todos os pontos de verificação particulares, para poder trilhar o aninhamento de comunicações e, assim, atualizar devidamente as informações manipuladas pelo gerente de recuperação; salvo se o estouro do "deadline" ocorrer em $S_3$. Piora a esquematização se for para atender o caso da figura VI.2 (c), salvo quando a detecção do estouro do "deadline" for feita em $S_3$, pois, como no caso anterior, a recuperação seria feita na ordem do aninhamento.

Em ambas as alternativas de solução ( a (1) e a (2) ) para os casos ilustrados pelas figuras VI.2 (b) e (c), elas não são viáveis, salvo para aquela situação particular; detecção do estouro do "deadline" em $P_4$.

Observa-se ainda, que como as três primitivas compõem a forma de comunicação por meio do "rendez-vous" estendido e as medidas para tolerância a falhas foram tomadas pelas recebe_msg_rendez_vous e envia_resposta_rendez_vous, a envia_msg_rendez_vous não deverá executar nenhuma operação nesse sentido. Esse esquema segue o princípio de "é melhor tratar a falha tão logo ela seja detectada, para evitar que seus efeitos se propaguem".

## C.2) TÉRMINO ANORMAL DE ENVIO DE MENSAGEM DEVIDO AO COLAPSO DO PROCESSO RECEPTOR

O Rajdoot não trata esse tipo de término anormal de chamada remota de procedimento. Considera que o colapso do servidor acaba por si próprio destruindo tudo que fez desde o último ponto de verificação e se ele realizou operações não idempotentes, é tarefa do processo ( cliente ou servidor ) tomar as devidas providências.

Para o presente mecanismo de comunicação e sincronização é interessante averiguar que medidas ele pode tomar sem que entre em conflito com aqueles para recuperação de processos. Dessa forma, primeiro é necessário definir, pelo menos, uma forma de como será detectado tal evento. Pela bibliografia estudada e já referida, o colapso de processo ou de nodo operador ( ou de comunicação ), pode ser detectado por:

(1) cão de guarda: um processo que de tempo em tempo, verifica se os nodos operadores estão em pleno funcionamento. No caso essa tarefa fica relegada para o processo gerente de recuperação;

(2) cada nodo operador de tempo em tempo envia uma mensagem informando que ele está em pleno funcionamento a algum supervisor. No caso essa mensagem deve ser dirigida ao processo gerente de recuperação;

(3) cada processo em plena ordem tem a responsabilidade de enviar de tempo em tempo uma mensagem a algum supervisor; no caso ao gerente de recuperação.

Independentemente, no momento, do método, a detecção de falha de processo servidor deve ser feita no menor tempo possível, a fim de impedir que seus efeitos se propaguem e fujam de controle.

Diante da presença de falha do processo servidor, o processo cliente que estava suspenso à espera de uma resposta, pode ser novamente ativado por estouro de tempo limite, ou pelo próprio mecanismo de recuperação. Caso ele seja sinalizado por estouro do tempo limite, os caminhos que o processo emissor da mensagem pode tomar são aquelas descritas em (i), (ii), e (iii). Assim, como descrito, é interessante que ele não tome medidas de recuperação, deixando-as para o processo servidor ou para as primitivas pares de comunicação, ali necessárias. Já na outra situação, se verá a seguir.

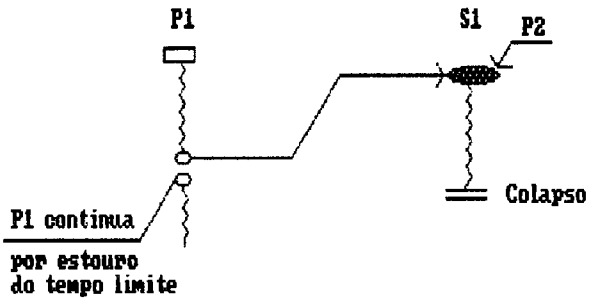



Figura VI.6: Término anormal de comunicação por colapso do processo servidor

## ALTERNATIVAS DE SOLUÇÃO

(1) Admitindo-se a hipótese de que o gerente de recuperação é avisado de que um processo ( ou todos de um nodo operador ) entrou em colapso, antes que o tempo limite de envio da mensagem pelo processo cliente tenha terminado:

- O gerente de recuperação verifica se o processo é ou não recuperável;
- se for recuperável, então, terá que avisar seus clientes ( feito via tabela de conexões de portos ), congelar seus tempos limites e recuperar o processo normalmente; ou
- se não for recuperável, então, o gerente de recuperação procede da mesma forma que para processos recuperáveis até o instante de congelar os tempos limites, inclusive. Em seguida,

atualiza as informações do cliente de que determinados processos com quem ele está em comunicação, não estão mais funcionando.

(2) Admitindo-se a hipótese de que o gerente de recuperação é avisado de que um ( ou todos de um nodo operador ) processo entrou em colapso, depois que o tempo limite de envio da mensagem pelo processo cliente tenha terminado:

- Para esse caso, fica um tanto complexo definir um esquema de recuperação, considerando que o processo cliente poderá se decidir por tomar qualquer caminho. Por exemplo, aquelas citadas em C.1 (1) ou (2), justamente pelo mecanismo de recuperação considerar todas as mensagens já enviadas e recebidas e ele acontecer após o processo cliente já estar em processamento e, assim, estar em algum outro estado não mais em sintonia com a recuperação executada. Dessa forma, por questões de viabilidade, deve-se desconsiderar essa hipótese, ou seja, quando da ocorrência de colapso do nodo operador onde reside o processo servidor em pauta, a única hipótese que pode acontecer será a anterior.

Nota-se que a alternativa (1) é válida para as seguintes formas de cooperação: figura VI.2 (a) e figura VI.3 (a). Para as demais formas ela não é viável e para maior percepção da complexidade da elaboração de uma, será feita uma análise mais adiante ( item C.4 ), abordando os problemas de computações órfãs para o caso da figura VI.3 (c).

## C.3) A COMUNICAÇÃO É INTERROMPIDA POR COLAPSO DO PROCESSO EMISSOR

Para esse caso, o esquema de Rajdoot garante que quando o cliente se recuperar, todas as computações órfãs geradas são abortadas ou pelo próprio mecanismo de comunicação ou pelo processo exterminador. Lembre-se, entretanto, que ele apenas aborta as computações em andamento devido ao colapso do nodo cliente. Deixa para os processos a tarefa de desfazer operações se for necessária.

Diante da hipótese de que o envio de uma mensagem sempre chega ao seu destino, desde que ele não esteja em colapso, a primitiva envia_resposta_rendez_vous tem plena condição de detectar esse evento

indesejável. Dessa forma, as alternativas de solução que seguem são baseadas na forma de comunicação da figura VI.7.

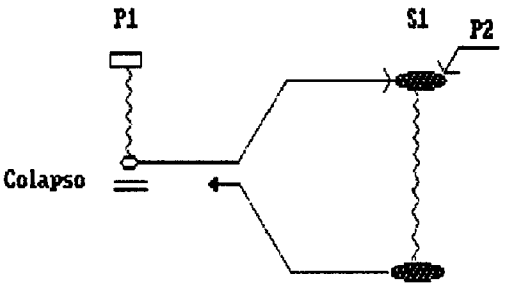



Figura VI.7: Comunicação entre processos afetada pelo colapso do processo cliente.

## ALTERNATIVAS DE SOLUÇÃO

(1) A envia_resposta_rendez_vous se comporta de forma idêntica à da alternativa C.1.1 (1);

(2) A envia_resposta_rendez_vous desfaz a computação como no caso por estouro de "deadline" ( C.1.1 (2) );

(3) Admitindo-se que o processo cliente seja recuperável, o processo servidor pode adotar a computação e ficar preso até a recuperação daquele, quando, então, o gerente terá que sinalizá-lo, para permitir a continuação de seu processamento que deverá ocasionar o envio da resposta;

(4) Admitindo-se que o processo cliente seja recuperável, o processo servidor adota a computação e avisa o gerente de recuperação para que ele atualize o contador de mensagens já enviadas e recebidas da cópia do emissor, para que a mensagem desta seja considerada como uma nova. Em seguida, se coloca à espera de mensagem. Quando essa chegar, ele simplesmente envia a resposta, que estava devidamente salva;

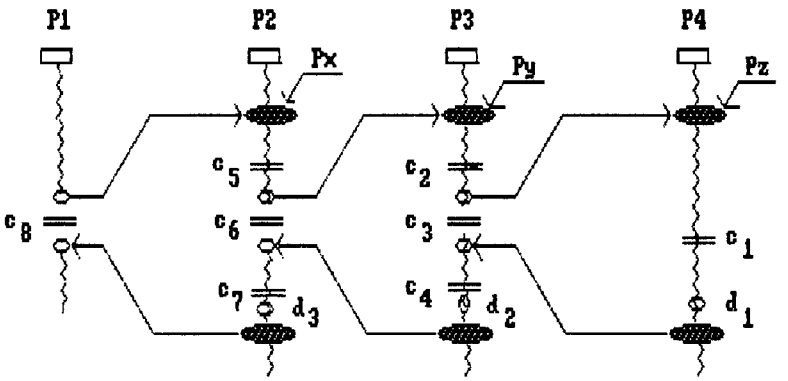
(5) Caso o processo cliente não seja recuperável, pode-se usar a alternativa (1), se a computação realizada for idempotente, caso contrário, pode-se usar a alternativa (2), atualizando apenas as informações da cópia do processo servidor.

Observe que a alternativa (3) é muito restritiva, no sentido de não permitir que outros processos que queiram se comunicar com esse processo servidor o façam e dêem continuidade a seus processamentos. Já a alternativa (4) não impõe essa restrição, entretanto, aumenta o

tempo de processamento da recebe_msg_rendez_vous, pois, a cada mensagem que recebe, ela deve verificar se esta é daquele processo cliente que já a enviou e em seguida entrou em colapso. Porém, ganha o tempo de processamento da extensão do "rendez-vous". Agora, a alternativa (2) é genérica. Ela simplesmente desfaz o que foi feito por não ter conseguido enviar a resposta devido o processo cliente estar em colapso e, fica pronto para receber novas mensagens e de quaisquer processos. A alternativa (1) é válida para o caso de computações idempotentes. Por último, as alternativas (5), são semelhantes às (1) e (2).

Como as duas primeiras alternativas, são idênticas às aquelas citadas, elas são válidas para as mesmas formas de cooperação entre processos, como também a (5), por ser uma particularidade daquelas.

## C.4) ANÁLISE DE FALHAS DE COMUNICAÇÃO PARA O CASO GENÉRICO DE COOPERAÇÃO ENTRE PROCESSOS




P - processos ( apenas cooperantes ou do tipo servidores )
d - estouro de "deadline"
c - colapso

Figura VI.8: Uma forma genérica de cooperação entre processos e pontos passíveis de falhas.

HIPÓTESES:

- Os mesmos feitos até então a respeito de término anormal de envio de mensagem ou impossibilidade de enviar mensagem de resposta;

- Pela figura VI.8, se $P_2$ é do tipo servidor, então, os seguintes também o são;
- Pela figura VI.8, se P3 é do tipo servidor, então, o seguinte também o é, e se P2 for processo cooperante, P1 também o é;
- a mesma que a anterior, porém, se P2 for do tipo servidor, P1 pode ser tanto servidor como processo cooperante;

**ANÁLISE DA OCORRÊNCIA DO ESTOURO DE "DEADLINE"**

(1) Se ocorreu $d_1$, as soluções são aquelas já descritas;

(2) Se ocorreu $d_2$ ou $d_3$, nota-se que, alguma solução se mostra bastante complicada. Por exemplo, se ocorreu $d_2$, a computação de $P_4$ devido ao envio da mensagem por $P_3$, já está sendo utilizada, uma vez que para $P_4$, a extensão do "rendez-vous", terminou normalmente.

**ANÁLISE DA OCORRÊNCIA DE COLAPSO**

(1) Se ocorreu $c_1$, as soluções são aquelas apontadas em C.2;

(2) Se ocorreu $c_2$, significa colapso de processo receptor para $P_2$. Se P2 não tivesse enviado a mensagem dentro de uma extensão de "rendez-vous", uma solução seria aquela apontada em C.2.1. Entretanto, como ele o fez, o problema grave que surge é o seguinte, na tentativa de usar a mesma solução: como congelar o tempo disponível para o término da comunicação de P1 ( no caso )?. Para isso, seria necessário o conhecimento do aninhamento dos envios de mensagens. Uma alternativa seria P3, quando recuperado, desprezar a mensagem que, como conseqüência, teria que solicitar que P2 também desprezasse a mensagem que originou esta e assim por diante, até atingir o processo emissor originador do encadeamento de envios de mensagens, no caso P1. Essa alternativa seria válida apenas para computações idempotentes e com as devidas sinalizações para o gerente de recuperação atualizar suas informações corretamente.

Dessa forma, já dá para perceber a complexidade de eliminação de computações órfãs, independentemente se elas são ou não idempotentes. Porisso, não se descreverão as análises para os demais casos, porém,

apenas a título de ilustração, se indicará como o colapso de um processo é visto pelos com quem ele está em comunicação.

(3) Se ocorreu $c_3$, significa colapso de processo receptor para $P_2$ e colapso de processo emissor para $P_4$.

(4) Se ocorreu $c_4$, significa colapso de processo receptor para $P_2$ e pode ter deixado órfão.

(5) Se ocorreu $c_5$, significa colapso de processo receptor para $P_1$.

(6) Se ocorreu $c_6$, significa colapso de processo receptor para $P_1$ e colapso de processo emissor para $P_3$.

(7) Se ocorreu $c_7$, significa colapso de processo receptor para $P_1$ e pode ter deixado órfãos.

(8) Se ocorreu $c_8$, significa colapso de processo emissor para $P_2$.

Objetivou-se mostrar com isso, a complexidade do tratamento de computações órfãs, quando a comunicação é feita segundo o esquema ilustrado pela figura VI.8. Porém, é interessante verificar se tal tipo de comunicação é muito utilizado, se ele reflete as pretensões de um programa distribuído e que qualquer outra forma alternativa acabar prejudicando a sua legibilidade textual e, porisso, vale a pena a construção de mecanismos para tratar órfãos que resolvam esse caso também. A princípio, percebe-se que, os tipos de comunicação mais visíveis de serem utilizados com frequência, são aqueles ilustrados nas figuras VI.2 e figura VI.3 (a). Assim, se ignorará esse caso genérico, como também, o ilustrado pela figura VI.3 (b), para a definição dos mecanismos para a detecção e eliminação de computações órfãs, para o mecanismo de comunicação e sincronização de processos baseado em portos.

## VI.4 USO DE PORTOS PARA COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS COM DETECÇÃO E ELIMINAÇÃO DE ÓRFÃOS, INTEGRADO COM O SUPORTE PARA RECUPERAÇÃO E MIGRAÇÃO DE PROCESSOS

Esse item se constitui na contribuição da tese. Essencialmente consiste da definição de um mecanismo de comunicação e sincronização entre processos baseado em portos com detecção e eliminação de órfãos, integrado com o suporte para os mecanismos para recuperação e migração de processos.

Nos itens anteriores, mais precisamente os VI.1 e VI.2, justificara-se a necessidade de um suporte para a recuperação e migração de processos e da opção pelos esquemas adotados para tanto, como também, observações a seu respeito, visto que eles não poderiam ser utilizados diretamente como propostos. Em seguida, analisaram-se no item VI.3, as possibilidades do surgimento de órfãos ( computações órfãs, para ser mais exato ) quando da utilização de mecanismo para comunicação e sincronização entre processos baseado em portos e foram propostas soluções, supondo a existência do suporte mencionado.

Assim, a seguir, descrevem-se esses mecanismos, ou seja, o mecanismo para recuperação e migração de processos e o mecanismo para comunicação e sincronização entre processos baseado em portos, com suporte para detecção e eliminação de órfãos.

## VI.4.1 MECANISMO DE RECUPERAÇÃO E MIGRAÇÃO DE PROCESSOS

A arquitetura base do Sistema Distribuído, para o qual se descreverá o mecanismo de recuperação e migração de processos e, em seguida, o mecanismo para comunicação e sincronização entre processos, é a do tipo ilustrado pela figura VI.9.

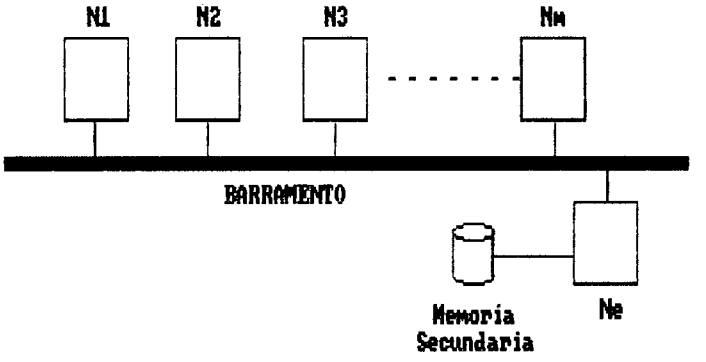



Figura VI.9 Arquitetura do Sistema Distribuído

Onde:

- N1,N2,...,Nm são os nodos operadores;
- Ne é um nodo operador especial, no sentido de dever estar superprotegido e que deverá ter uma boa capacidade de memória e um dispositivo de memória secundária de alta capacidade e de alta velocidade. Esse nodo deverá ter essas características porque será

utilizado como o nodo operador responsável pela recuperação e migração de processos, como também, responsável pela configuração inicial do Sistema Operacional Distribuído.

- Todos esses nodos operadores devem estar interligados por um barramento. No caso, se admitirá que esse elemento do Sistema Distribuído seja tolerante a defeitos, ou seja, a presença de um, não deve implicar no isolamento de nodos operadores, como ilustrado na figura VI.10.

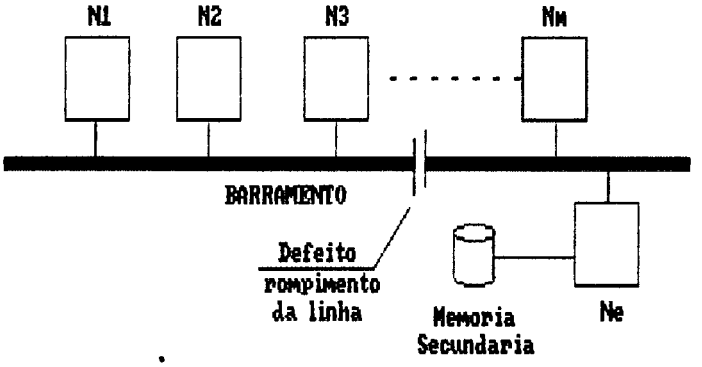



Figura VI.10: Um problema gerado pela ocorrência de rompimento do meio de comunicação entre nodos operadores.

Se admitirá, por enquanto, a existência em cada nodo operador de um núcleo que oferece um serviço de comunicação entre nodos operadores e de alguns outros, que mais adiante se esclarecerá.

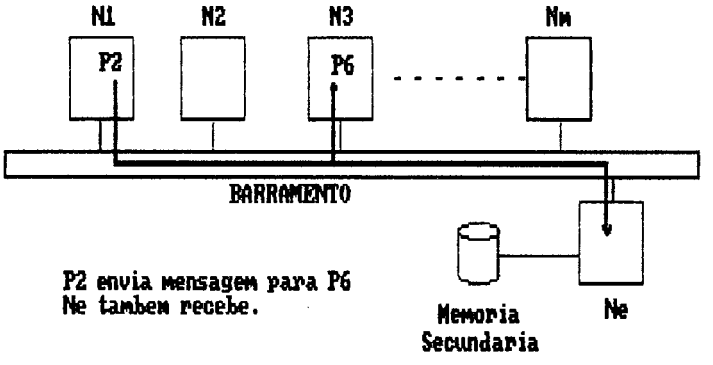



Figura VI.11: Esquema do funcionamento da transmissão de uma mensagem de um nodo operador para um outro.

A função principal desse serviço de comunicação entre nodos operadores é a de transmitir uma mensagem solicitado por algum processo para um outro, também para aquele nodo operador especial

responsável pela recuperação e migração de processos ( veja figura VI.11 ).

Como delineado no item IV.3.1, o mecanismo de recuperação de processos, consiste basicamente de um processo gerente de recuperação e um banco de dados. O banco de dados, consiste das cópias dos processos recuperáveis e respectivos estados. Cada estado de um processo, contém todas as informações necessárias para a recuperação de seu original, a partir do último estado salvo, determinado quando do estabelecimento de ponto de verificação, como mais adiante será esclarecido.

No presente mecanismo de recuperação de processos, a tarefa de determinar quando se deve estabelecer um ponto de verificação em cada processo recuperável, será delegada ao próprio, isto é, ao seu processo gerente de recuperação de processos. Também, como mencionado no item VI.1, o gerente de recuperação terá a tarefa de executar a migração de processos, devido esta, ser um caso particular de recuperação de processo. Porisso, se denominou esse processo gerente, de gerente de recuperação e migração de processos, que daqui por diante será referenciado simplesmente por GRMP.

A tarefa de criação de processos de usuários, ficará aos cuidados dos nodos N1, N2,..., Nm. Cada um criando os processos de seus usuários locais físicos. A tarefa de iniciar o Sistema Operacional será de responsabilidade do Ne, como também a primeira alocação para execução de processos de usuários, como no esquema análogo ao desenvolvido em OLIVEIRA ( 1991 ). Isso diante da aderência dessa técnica, para o presente tipo de arquitetura, ser bastante grande, pois a principal diferença está em que, enquanto que aquele é uma hierarquia, onde todos usuários estariam conectados na raiz, neste, a hierarquia seria sendo estabelecida pelo "software".

Resumidamente, a técnica é a seguinte: cada novo processo que entra no sistema, é definido pelo Sistema de Balanceamento Distribuído de Carga ( residente no nodo raiz ), o nodo operador menos sobrecarregado, é para onde, então, o processo recém criado, irá ser alocado e processado. Obviamente, a tarefa de criar os processos cópias ficará para o Ne, que é o responsável pela sua manutenção. Agora, quanto quando criar, nota-se que tanto pode ser a pedido de cada núcleo de cada nodo operador, ou como esse nodo Ne já está ciente da existência de um novo processo e onde ele deverá ser alocado, poderia, nesse instante, executar tal função. Pelo atual estágio em que se

encontra a parte implementada do projeto em questão, esses problemas serão postergados, ciente de que eles não prejudicarão o progresso do mesmo. Assim sendo, o Ne deverá criar processos cópias a pedido de N1, N2,... e Nm, independente de quando.

Como explicitado no item VI.3.1, existem várias formas para se detectar o colapso de um processo ou de um nodo operador: pelo cão de guarda; pelo silêncio de um núcleo; e pelo silêncio de um processo. Considerando que a escolha por qualquer um deles deve ser feita sob a ótica de várias variáveis, por exemplo, o tipo da rede de comunicação entre nodos operadores, a velocidade de transmissão, tipos de aplicações a que destina o Sistema Distribuído, etc., fica difícil a opção pura e simples por alguma delas. Considerando que, a utilização de uma ou de outra técnica, ou até uma que seja a mescla de algumas ou de todas, pode influir apenas em alguns pontos bem determinados dos mecanismos objetos do presente trabalho. Optou-se por definir uma mescla do silêncio de núcleo com o cão de guarda, pelos seguintes motivos:

- o colapso de um processo só poderá ocorrer se houve o colapso de seu nodo operador ( veja item VI.3.1. B ); e
- julga-se que diante da possibilidade da existência de mais do que um processo em cada nodo operador, a probabilidade de qualquer um deles necessitar se comunicar com um outro ( local ou remoto) é grande.

O que leva a crer que raramente um núcleo ficará em silêncio por muito tempo, salvo se estiver em colapso. Em síntese, essa técnica será a seguinte:

- o GRMP contabilizará através do fluxo de mensagens entre processos, quais estão em silêncio por um determinado intervalo de tempo;
- caso todos os processos residentes em um nodo operador estejam em silêncio, o GRMP envia uma mensagem para o núcleo deste nodo, para averiguar se ele realmente está ativo.

Note que, à primeira vista, essa técnica mostra apenas as suas virtudes, pois ela exigirá o mínimo de sobrecarga sobre o recurso do Sistema Distribuído mais compartilhado, que é a rede de comunicação entre nodos operadores. Entretanto, por outro lado, ela sobrecarrega o Ne. Porém, esse nodo terá somente as tarefas de recuperação e migração de processos e de configuração do sistema, não compartilhando o seu processador com nenhum processo de usuários.

Conseqüentemente, acredita-se que ele não imporá uma queda no desempenho do sistema ao exercer mais essa tarefa.

Antes de descrever as outras funções que o mecanismo de recuperação e migração de processos deve exercer, colocam-se as hipóteses sobre as quais ele está baseado, que são as seguintes ( algumas já foram estabelecidas ):

(1) o nodo operador Ne, não executará processos de usuários. Terá apenas as funções de executar o gerenciamento de recuperação e migração de processos e outras, como já citadas, a de executar a configuração inicial do Sistema Operacional Distribuído, etc;

(2) os nodos operadores N1, N2, ..., Nm e Ne são idênticos. Dessa forma o mecanismo de migração de processos, por questões de recuperação ou de balanceamento de carga ou etc, não precisará se preocupar em executar a translação de código de processo ( veja item IV.4 );

(3) o barramento que interliga os nodos operadores é tolerante a defeitos, como já estabelecido;

(4) o nodo Ne deve estar bem protegido contra queda de energia e contra o seu desligamento por acidente, como também, deve apresentar uma confiabilidade bem maior que os outros nodos operadores. Isso porque, ele se constitui, a princípio, no único ponto vulnerável do Sistema Operacional Distribuído. Assim, para maiores esclarecimentos, por exemplo, o nodo, quando da percepção de queda de energia, poderia por meio de interrupção ( "power fail" ), acionar uma rotina para permitir a restauração do Sistema Operacional a partir do instante da ocorrência desse evento; ou permitir que o operador do Sistema possa requisitar a continuação do processamento, ou etc.. Esse é um assunto que carece de maiores estudos. Para o caso de sistema centralizado, o esquema de "power fail" já é bastante utilizado. Entretanto, como no presente tipo de sistema, os componentes tanto de "hardware" como de "software" estão distribuídos e têm uma certa autonomia, percebe-se que esse esquema não poderá ser utilizado de forma exatamente igual, como também, com pequenas adaptações. Quanto ao problema de confiabilidade, a sua importância está em que esse nodo é o único que conterá a história de cada processo. Conseqüentemente, sua perda implicará na perda de informações para as suas recuperações. Esse é um problema, que se nota não

ser muito complexo de ser resolvido. Entretanto, não será aqui abordado;

(5) o dispositivo de memória secundária de alta capacidade e de alta velocidade conectado ao Ne, é para o armazenamento das cópias dos processos ( os "back-ups" ) e respectivos estados. A característica de alta capacidade é auto justificada. Já a característica de velocidade está relacionada ao desempenho do sistema. Quando maior for a velocidade, ou melhor, quanto menor for o tempo de acesso a esse meio de armazenamento de informações, menor será a queda do desempenho do sistema;

(6) o salvamento do estado de um processo para fins do estabelecimento de um ponto de verificação, não deverá ocorrer quando ele estiver executando alguma ação atômica, pois, iria contra a definição desta, caso ocorra a necessidade da recuperação neste ponto.

Feito isso, passa-se a seguir, a resumir os principais procedimentos do mecanismo para recuperação e migração de processos ( o GRMP ), sem a preocupação quanto à ordem de execução:

(1) O GRMP ao receber uma mensagem enviada por um processo para um outro, que não seja somente para ele, deve tomar as seguintes providências:

(a) verificar a validade da mensagem no sentido de que se realmente existem os processos cópias. Caso existam, então, enviar a confirmação de recebimento da mensagem ao mecanismo de comunicação entre nodos operadores, para que o processo emissor possa continuar com as suas tarefas. Caso contrário, comunicar esse fato, ao processo emissor, para que ele possa tomar as devidas providências;

(b) armazenar a mensagem no "buffer" de recepção de mensagens do processo cópia destino; e

(c) incrementar o contador de mensagens já enviadas e recebidas da cópia do processo emissor da mensagem.

(2) a cada determinado intervalo de tempo ou a detecção de que um processo enviou e/ou recebeu 10 mensagens, é solicitado ao núcleo do nodo operador onde reside aquele processo, o estabelecimento de um ponto de verificação;

(3) o GRMP ao receber a resposta de (2), deve atualizar o estado do processo cópia, descartando dele as mensagens recebidas e zerando o contador de mensagens já enviadas e recebidas;

(4) o gerente de recuperação deve conter um "buffer" para ir armazenando as mensagens que chegam enquanto ele estiver executando alguma tarefa;

(5) o gerente de recuperação tem acesso às informações sobre a configuração do sistema, como também as capacidades de cada nodo operador, como dispositivos periféricos, tamanho de memória e a parte disponível ( caso o nodo não possua memória virtual ), e etc., para justamente poder tomar decisões seguras, quando ele deve executar algum serviço. Por exemplo, deve ser capaz de detectar a incapacidade de recuperar um processo recuperável, devido a falta de recursos físicos ( falta de espaço de memória, por exemplo );

(6) quando o GRMP perceber que algum nodo operador está em silêncio, após decorrer um determinado intervalo de tempo, ele deverá enviar uma mensagem ao seu núcleo, solicitando que envie um sinal de vida. Caso chegue tal confirmação, então, está tudo bem. Caso contrário, o GRMP deverá executar a tarefa de recuperação dos processos ali residentes, da seguinte forma:

   (a) por meio de tabela, ele verifica quais desses processos não são recuperáveis e quais são;

   (b) para os processos recuperáveis, com o auxílio de informações a respeito da carga de cada nodo operador e de outros fatores, por exemplo as taxas de comunicação entre processos, para procurar colocar os processos com altas taxas em um mesmo nodo operador, ele determina os nodos para onde estes processos irão ser recuperados. Feito isso, o GRMP atualiza todas as informações do sistema para refletir a sua nova configuração, como as tabelas de conexões de portos, tabela de localização dos processos, etc. Em seguida, solicita aos núcleos desses nodos operadores a reserva de espaço de memória e que fiquem à espera da emissão dos códigos dos processos e de seus respectivos estados com a marcação de que estes estão em recuperação, por parte do GRMP. Tal marcação, permite que os processos que queiram se comunicar com eles fiquem cientes desse fato.

   (c) ainda para os processos recuperáveis, com o auxílio das tabelas de conexões, ele fica sabendo com quem esses processos estão em comunicação. Assim, o GRMP tem a capacidade de avisar

esses processos que seus emissores de mensagens estão em recuperação;

(d) para os processos não recuperáveis, o GRMP deve avisar os processos com que ele se comunica, desse fato, para que eles possam tomar suas devidas providências;

(e) vale o mesmo procedimento que (d) caso não exista recurso suficiente nos nodos ativos;

(7) quando um processo estiver em recuperação e ocorrer o colapso do mesmo, o gerente deve ser capaz de detectar e reiniciar a recuperação;

(8) o gerente de recuperação deve ser capaz de detectar mensagens duplicadas;

(9) o gerente de recuperação pode receber mensagens enviadas diretamente a ele. Tais mensagens devem ser comandos para que ele execute algum serviço. Para o momento, a migração de um processo e a atualização de informações de processos cópias ( descartar mensagem e/ou decrementar contador de mensagens enviadas e recebidas, emitido pelo mecanismo de tratamento de órfãos ). Os passos para a execução da migração de um processo são os seguintes:

(a) suspender a execução do processo a ser migrado, em algum ponto permitido;

(b) marcar o processo a ser migrado, suspendendo-o na fila dos processos suspensos por esse motivo;

(c) estabelecer um ponto de verificação;

(d) atualizar as informações sobre a nova localização do processo em todo o sistema;

(e) solicitar ao núcleo do nodo operador para onde o processo deve migrar, que crie um processo vazio ( criar um descritor de processo ) com a mesma identificação que aquele, uma pilha de trabalho e outras informações necessárias;

(f) solicitar ao núcleo do nodo operador para onde o processo deve migrar, que receba as informações sobre o processo a ser ali migrado do GRMP, para o preenchimento das estruturas de dados que refletem o estado completo do processo;

(g) ativar o processo migrado, colocando-o como pronto para execução;

(h) ativar todos os processos que estejam suspensos por esse motivo;

(i) uma observação: para evitar a criação do processo roteador, como utilizado em DEMOS/MP, se implementará o núcleo de cada nodo operador com as funções de, ao detectar que a mensagem recebida é para um processo que não mais existe, enviar como resposta tal informação e, que ao receber esse tipo de resposta sinaliza o mecanismo de comunicação de nível logo superior desse evento que, por sua vez, deverá tomar as devidas medidas que são: consultar novamente as tabelas de conexões de portos para obter o novo endereço; e reemitir a mensagem para o novo endereço. A possibilidade de as tabelas de conexões não estarem ainda atualizadas, é descartada ao se priorizar tais tipos de mensagens.

Dessa forma, o algoritmo esquemático do processo GRMP é o seguinte, o qual, praticamente, apenas ordena a seqüência da execução dos procedimentos acima listados:

```
repita
    (* ------------------- PARTE  A --------------------------------
      parte do gerente que recebe as mensagens em trânsito, analisa-
      as e toma as devidas providências. A recepção dessas mensagens
      é feita de forma bloqueada, por meio de PortoGerente
      ------------------------------------------------------------ *)
      receber_msg ( PortoGerente, M, ... );
      mf := identificação_da_fonte ( M );
      md := identificação_do_destino ( M );
      me  := mensagem ( M );
      se md é o GRMP
      então executar comando especificado em me
          me = migrar ==> executar a migração de processo;
               (* esse comando é solicitado pelo mecanismo de
                  balanceamento de carga
               *)
         me = atualizar inf. dos proc. cópias ==> executar esses
                                                  serviços;
               (* esse comando é solicitado pelo mecanismo de
                  tratamento de órfãos
               *)
```

```
senão
   se md e mf existem
   então
       atualizar inf. dos proc. cópias;
       se mf enviou mais do que 10 mensagens
       então
          emitir solicitação de marcação de ponto de
                  verificação;
          esperar resposta;
          atualizar inf. do proc. cópia;
       fim_se;
       se md recebeu mais do que 10 mensagens
       então
          emitir solicitação de marcação de ponto de
                   verificação;
          esperar resposta;
          atualizar inf. do proc. cópia;
       fim_se;
   senão enviar como resposta esse problema;
   fim_se;
fim_se;

(* ------------------ PARTE B --------------------------------
 essa parte é para verificar se algum nodo entrou em colapso ou
 todos os seus processos não se comunicam com algum outro
 remoto
   ------------------------------------------------------------ *)
se passou o intervalo de silêncio
então
   enquanto existir algum processo que não enviou
             ou não recebeu nenhuma mensagem
   faça
       enviar mensagem ao núcleo do nodo operador
          destino solicitando a marcação de um ponto de
          verificação;
       esperar resposta;
       se não chegou resposta
       então
           marcar os processos cópias que serão
```

```
                utilizados para a recuperação de seus
                originais;
            iniciar a tarefa de recuperação dos processos;
        senão
            atualizar inf. do proc. cópia;
    fim_enquanto;
fim_se;
para sempre;
```

Cabe observar que a PARTE B poderia ser esquematizada de forma que, quando encontrasse um processo que ficou em silêncio pelo tempo maior que o pré-determinado intervalo de tempo, executasse a tarefa ali resumida. Isso permitiria um maior paralelismo entre essa tarefa e a PARTE A. Entretanto corre-se o risco de atrasar demais a detecção do colapso de algum nodo operador. Assim, optou-se pela apresentada.

Uma outra observação é quanto à implementação do algoritmo todo. Poderia-se implementar as PARTE A e PARTE B, como duas rotinas de tratamento de interrupções. Isto é, a PARTE A seria ativada toda vez que chegasse uma mensagem proveniente do meio de comunicação entre nodos operadores e, a PARTE B, quando o relógio de tempo real de Ne marcasse o término do intervalo de tempo especificado. No presente, está admitido que o relógio de tempo real apenas serve para incrementar ou decrementar a variável, representante do intervalo de tempo.

### VI.4.2 MECANISMO PARA COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS BASEADO EM PORTOS COM DETECÇÃO E ELIMINAÇÃO DE ÓRFÃOS.

Objetivando que esse tipo de mecanismo de comunicação entre processo possa servir para Sistemas Operacionais Distribuídos de Uso Geral, optou-se, a princípio, em definir todos aqueles tipos de portos conforme descrito em V.1.3, ou seja:

- porto de entrada;
- porto de saída e
- porto bidirecional,

com:

- comunicação síncrona;
- comunicação assíncrona e
- com a utilização de "buffers" tanto no processo origem como no processo destino.

e permitindo os seguintes tipos de interligações:

- um para um;
- um para vários;
- vários para um e
- vários para vários.

Além disso, o mecanismo apresenta as características de:

(a) tolerância a falhas; e

(b) detecção e eliminação de órfãos.

Como observado nos itens anteriores, a estruturação desse "software" deve ser feita em camadas. Uma no nível do núcleo com a responsabilidade de criar a máquina virtual para execução dos processos locais e a outra, em nível logo inferior, que terá a função de oferecer a transparência da rede para todos os processos do sistema que, juntas, criam uma única máquina à vista de seus usuários.

### VI.4.2.1 ESTRUTURAÇÃO DO "SOFTWARE" DE PORTOS

O "software" do mecanismo para comunicação e sincronização entre processos baseado em portos com suporte para detecção e eliminação de órfãos, a ser implementado, deverá permitir aos processos que os utilize, através de suas primitivas, se comunicarem com outros processos, segundo as formas: um para um; um para vários; vários para um; e vários para vários, e sem se preocuparem com as localizações dos parceiros dentro do Sistema Distribuído, como também, da existência do suporte subjacente citado e dos mecanismos para recuperação e migração de processos descrito em VI.4.1.

O "software" foi estruturado em duas camadas hierárquicas, a saber:

- **camada do "software" de portos:** que conterá a implementação das estruturas de dados de portos e suas primitivas para comunicação e sincronização entre processos a serem oferecidas para a camada superior e que utilizará como suporte para comunicação remota, primitivas oferecidas pela camada inferior que segue, como também de primitivas que se farão necessárias para tratar questões

relativas à recuperação de processos, descritas em VI.4.1 e que estão melhor detalhadas no item VI.4.2.5, logo adiante;

- **camada do "software" de rede**: deverá conter todas as operações primitivas e informações necessárias para dar à camada superior a transparência da rede de comunicação subjacente.


Figura VI.12: Estrutura hierárquica do "software" de portos com suporte para detecção e eliminação de órfãos

## VI.4.2.2 TIPOS DE PORTOS

Os tipos básicos de portos são:

- **de entrada**: que pode somente receber mensagens;
- **de saída**: que pode somente enviar mensagens
- **de difusão ou bidirecional**: que tanto pode enviar como pode receber mensagens.

Esses portos serão do tipo local, no sentido de que apenas os processos que os criarem terão direito de utilizá-los, como também de destruí-los. Isso, como discutido, permite que os processos se tornem funcionais; independentes do contexto em que eles estão inseridos, o que não acontece quando se usam portos globais. Naturalmente, como já observado, os identificadores de todos os portos criados pelos processos, devem ser visíveis à parte do sistema responsável pelo estabelecimento das conexões entre os portos, para que esta defina a rede de comunicação lógica dos processos componentes do sistema todo - configuração do Sistema Operacional.

Além disso, cada porto será caracterizado pelo tipo de mensagem que ele pode receber ou enviar. Isso oferece uma maior segurança e confiabilidade. Segurança porque o tipo da mensagem praticamente é

uma espécie de capacidade do porto. Confiabilidade porque o tipo da mensagem que um porto pode enviar ou receber, fica textualmente explícito no código do programa, tornando-o assim, bastante claro para poder verificá-lo e modificá-lo ( legibilidade ). No caso de uma linguagem de programação oferecer tal tipo de objeto, essa característica permitirá verificar, em tempo de compilação, se suas operações primitivas estão sintaticamente corretas ( se o tipo da mensagem utilizada é compatível com o permitido pelo porto usado ).

No presente trabalho, a linguagem usada para a implementação do "software" é a Módula-2, como já frisado. Essa linguagem não oferece qualquer objeto como construção própria, que se assemelha com o desejado. Entretanto oferece facilidades para tal, por meio de seu módulo biblioteca SYSTEM ( facilidades de baixo nível ), como também permite que se crie módulos desse tipo de forma natural - programação modular.

Quanto à utilização de "buffers", cujas funções estão descritas no item V.1.3, se usarão nos portos tanto para comunicação síncrona como para a assíncrona, da seguinte forma:

(a) para cada porto para comunicação síncrona, será associado um único "buffer", isto é, "buffer" capaz de reter uma única mensagem;

(b) para cada porto para comunicação assíncrona, será associado uma fila de "buffers".

Para comunicação síncrona, como já observado, não haveria a necessidade de se usar "buffer", nem no porto de saída como no de entrada correspondente. Porém, considerando que a implementação dos portos e de suas primitivas ficarão na camada de "software" de portos, sua utilização e de forma simétrica ( "buffer" tanto no porto fonte como no porto destino ), permite tornar claro a intenção de que eles fiquem independentes do "software" subjacente ( a camada de "software" de rede" ), responsável pela comunicação confiável remota.

Para a comunicação assíncrona, as mesmas justificativas se aplicam, e mais, a necessidade da utilização de "buffers" é bem visível, pois, um processo que a usa pode ser liberado, para dar prosseguimento ao seu processamento normal, tão logo a mensagem a ser enviada seja armazenada em um desses "buffers" locais.

Ciente de que, por maior que seja a quantidade de "buffer", pode ocorrer que em um instante qualquer ela fique esgotada, se visualizam duas possibilidades de comportamento de uma comunicação assíncrona, quando esse fato ocorrer, que são:

- a comunicação se torna síncrona: essa técnica visa a não perda de nenhuma mensagem;
- a comunicação continua assíncrona: para tanto, deverá ser tomada uma das seguintes decisões, ou as novas mensagens que chegam são simplesmente descartadas, ou as novas mensagens se sobrepõem sobre as mais antigas.

Para o presente trabalho optou-se pela não perda de nenhuma mensagem. Além disso, diante do objetivo de se implementar na camada de "software" de rede, um único mecanismo de transmissão remota síncrona, este pode levantar a exceção de que ele não conseguiu transmitir uma mensagem. Assim, o porto pode descartá-la liberando um "buffer", ou simplesmente passar essa informação para o processo. Para o presente, incluiu-se essa última alternativa, justamente para permitir ao processo usuário, definir sua melhor forma de tratamento.

Assim sendo, serão criados, no momento, oito tipos de portos de acordo com os tipos básicos e para atender, tanto as formas de comunicação ( síncronas ou assíncronas ), como também, as topologias de interligações desejadas. Quanto à tipificação dos portos referentes aos tipos de mensagens, por apenas representar um pequeno detalhe de implementação e que não influi nas metas deste item, será oportunamente observado, nos itens seguintes.

Os oito portos e suas semânticas são:

- PSDRA ( Porto de Saída Dúplex com Resposta Associada ): é um porto de saída síncrona, que terá a característica de ser bidirecional, no sentido de que para cada mensagem enviada, fica a espera de uma mensagem resposta que terá que chegar por este mesmo porto. O tempo que o porto fica bloqueado a espera de uma mensagem resposta à enviada, é limitado por um tempo máximo pré-fixado pelo seu usuário. Por o porto ser síncrono ( bloqueante ), possuirá apenas um "buffer" para reter a mensagem a ser enviada ou a mensagem resposta recebida.
- PEDRA ( Porto de Entrada Dúplex com Resposta Associada ): é um porto de entrada síncrona, que terá a característica de ser bidirecional, no sentido de que para cada mensagem que chega, ele deve enviar uma mensagem resposta por este mesmo porto. O tempo de espera por uma mensagem é limitado por um tempo máximo pré-fixado pelo seu usuário. De forma análoga ao tipo de porto anterior, este possuirá,

também, apenas um "buffer" para reter a mensagem que chega ou a mensagem resposta a ser enviada.

- PSDCA ( Porto de Saída Dúplex com Confirmação Associada ): Análogo ao PSDRA, onde a mensagem resposta é substituída pelo sinal de confirmação.
- PEDCA ( Porto de Entrada Dúplex com Confirmação Associada ): Análogo ao PEDRA, onde a mensagem resposta a ser enviada se resume em apenas um sinal de confirmação.
- PSS ( Porto de Saída Simplex ): é um porto de saída unidirecional Assíncrona. Sua semântica será a seguinte: assim que a mensagem a ser enviada estiver armazenada em um "buffer" da fila de "buffers" do porto, o processo que o estiver usando será liberado para dar prosseguimento ao seu processamento normal. Caso não haja mais "buffer" disponível, o processo que está usando tal porto deverá ficar suspenso a espera da condição da existência de um "buffer" vazio, ou da exceção de que a mensagem não foi possível de ser enviada ao seu destino.
- PES ( Porto de Entrada Simplex ): é um porto de entrada unidirecional síncrona, com apenas um "buffer" associado. Sua semântica é a seguinte: se o "buffer" estiver vazio, suspende o processo que o está utilizando até que uma mensagem seja depositada. Para evitar que um processo fique suspenso para sempre, será fixado um tempo limite de espera.
- PD ( Porto de Difusão ): é um porto de saída unidirecional assíncrono com semântica idêntica à do PSS. A decisão da criação de tal tipo de porto, como melhor será percebido, no item seguinte, foi apenas por questão de clareza na compatibilidade de interligações de portos.
- PR ( Porto de Recepção ): é um porto de entrada unidirecional síncrono com semântica idêntica à do PES. Vale a mesma observação feita para o PD.

### VI.4.2.3 TIPOS DE INTERLIGAÇÕES DE PORTOS

Como visto, o conceito de porto local, oferece um meio de comunicação indireta, que não é exigido, e também não é necessário se

ter, o conhecimento dos processos envolvidos. Para um processo enviar uma mensagem, basta que ele o faça através de uma primitiva de emissão de mensagens, onde se identificam o porto local utilizado para tal e outros parâmetros que se façam necessários, menos a identificação do processo receptor. Assim, para que uma comunicação aconteça, é necessário primeiro estabelecer a devida conexão entre os portos dos processos envolvidos.

As conexões devem ser realizadas sempre entre pares de portos, formados por um porto cuja função primeira seja de saída com um cuja função primeira seja de entrada. Dessa forma, e de acordo com as características funcionais de cada tipo de porto, as possíveis conexões são:

(1) Um porto PSDRA pode ser conectado a somente um porto PEDRA;
(2) Um porto PEDRA pode ser conectado a vários portos PSDRA;
(3) Um porto PSDCA pode ser conectado a somente um porto PEDCA;
(4) Um porto PEDCA pode ser conectado a vários portos PSDCA;
(5) Um porto PSS pode ser conectado a somente um porto PES;
(6) Um porto PES pode ser conectado a vários portos PSS;
(7) Um porto PD pode ser conectado a vários portos PR; e
(8) Um porto PR pode ser conectado a somente um PD.

Enfim, as regras básicas estabelecidas pelas definições dos diversos tipos de portos, para as interligações válidas dos portos são:

(a) PSDRA conectado com PEDRA: comunicação totalmente síncrona para a implementação de comunicação segundo o esquema de "rendez-vous" estendido;
(b) PSDCA conectado com PEDCA: comunicação totalmente síncrona para a implementação de comunicação segundo o esquema de "rendez-vous" simples;
(c) PSS conectado com PES: comunicação assíncrona ( emissor não bloqueante e receptor bloqueante ), para a implementação de comunicação onde o processo emissor não necessita de nenhuma confirmação de que o processo destino tenha ou não recebido a mensagem emitida;
(d) PD conectado com PR: comunicação assíncrona ( emissor não bloqueante e receptor bloqueante ), para a implementação de comunicação por difusão para vários processos.

Note que existem pequenas diferenças entre as conexões (a) e (b) e entre as conexões (c) e (d). Diferenças essas que tiveram como objetivo que elas expressem textualmente a forma de comunicação que

se deseja estabelecer, se é segundo o esquema de "rendez-vous" estendido ou não, e se é segundo uma comunicação assíncrona simples ou por difusão, respectivamentes; o que facilita em muito a verificação dos códigos dos processos, à custa de memória necessária. Note também, que não foi criado um par de portos específicos para comunicação por difusão total. Isso porque, a sua aplicação pode ser efetuada pelo par de portos PD e PR, apesar deste não permitir estabelecer uma comunicação totalmente assíncrona. Para tanto, poderia-se criar um porto de entrada assíncrona que poderia ser conectado a um ou mais portos do tipo PD. Entretanto, diante da quantidade de aplicações que exigiriam tal tipo de comunicação ser bastante pequena e do objetivo

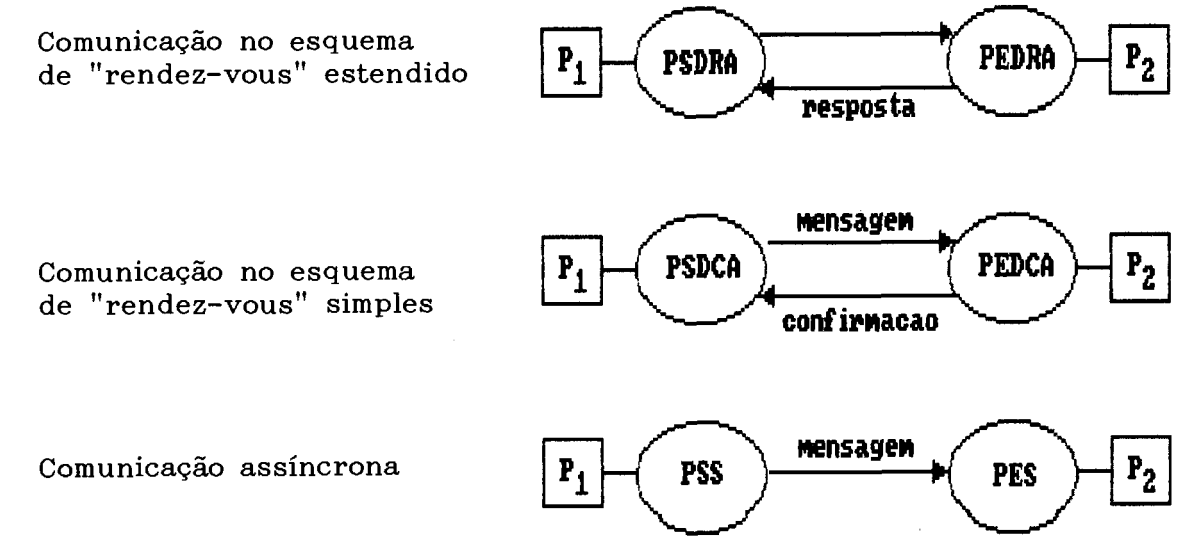



Figura VI.13: Interligações um para um

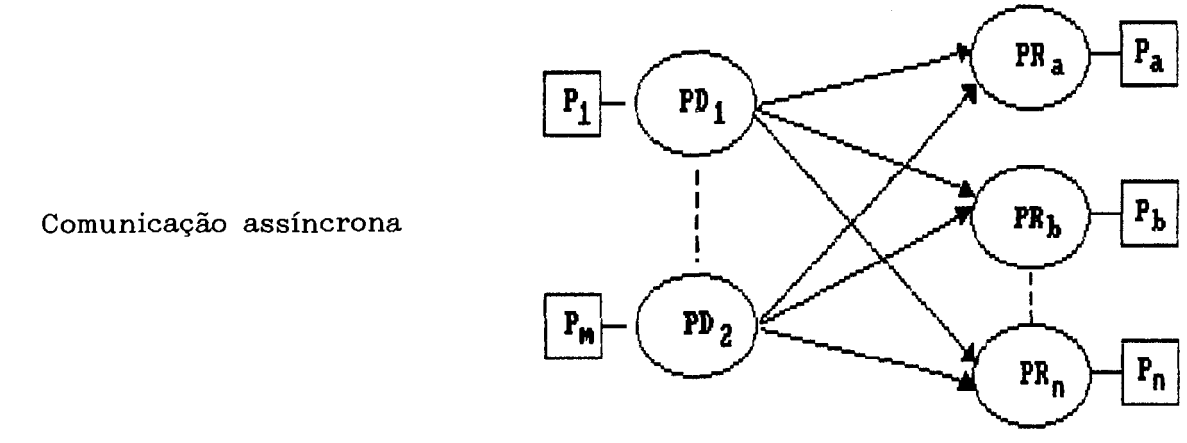



Figura VI.14: Interligações um para vários e vários para vários

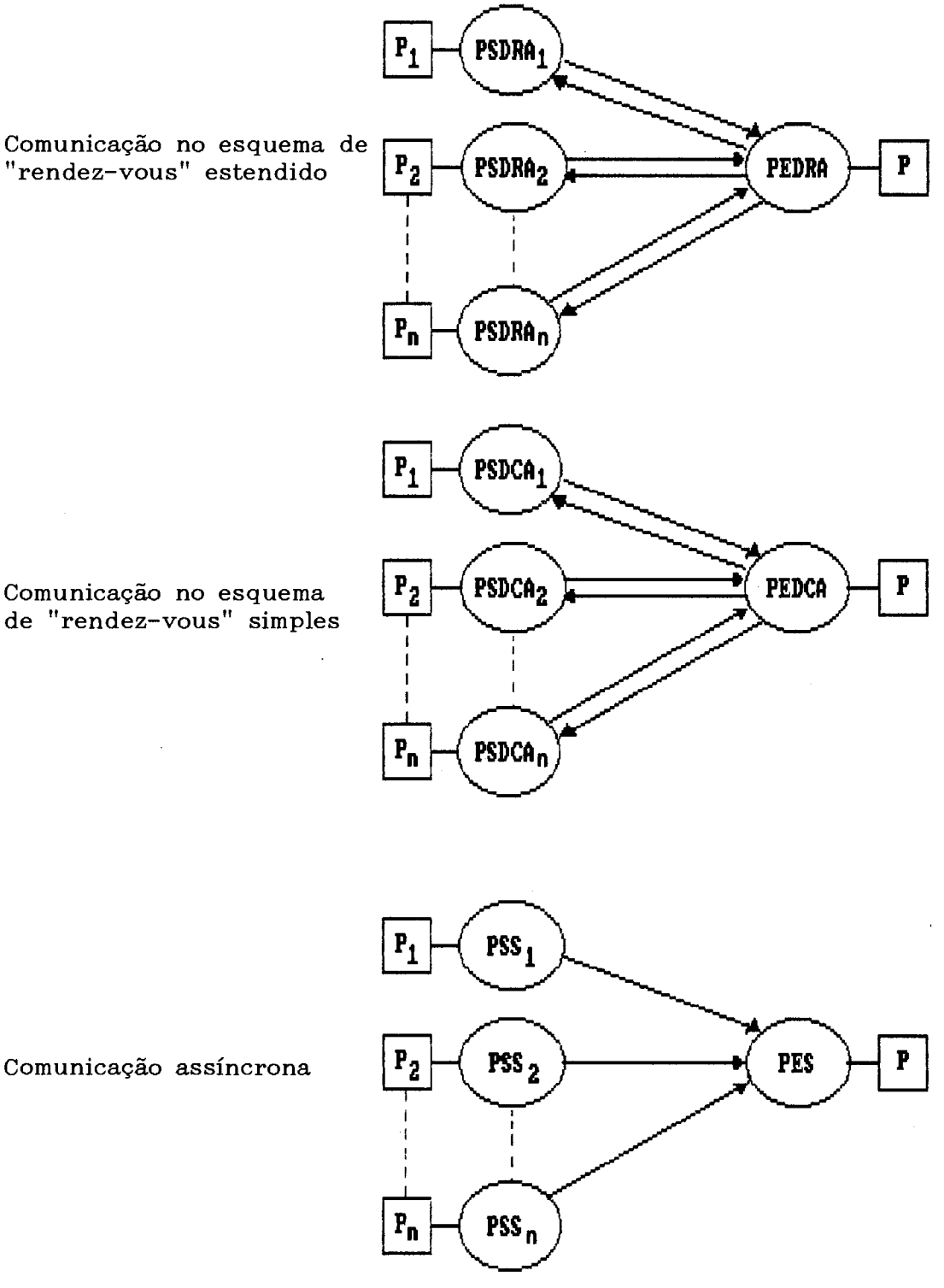



Figura VI.15: Interligações vários para um

perseguido de se colocar o máximo de confiabilidade no sistema, essa opção foi descartada.

Assim, percebe-se que os tipos de interligações de portos suportados são:

- um para um;
- um para vários;
- vários para um; e
- vários para vários, como uma combinação dos dois tipos anteriores.

As figuras VI.13, VI.14 e VI.15, ilustram os três primeiros tipos de interligações.

É importante que o estabelecimento das conexões possa ser feita como desfeita dinamicamente, o que permitirá que os processos possam migrar de um nodo operador para um outro. É importante também, como já ressaltado, que esse estabelecimento de conexão entre os portos não seja feito pelos próprios processos envolvidos, senão, cada um deveria conhecer os identificadores dos portos existentes e quem são seus donos, o que invalidaria o conceito de portos locais. Assim, essa tarefa será delegada a um único elemento do sistema, que será denominado de Configurador do Sistema, que terá também outras funções, como por exemplo, a de distribuir os processos pelos nodos operadores componentes do Sistema Distribuído. Essa forma centralizada facilita em muito a execução dessa tarefa, pois se tem a qualquer instante a atual configuração do sistema, definida pelos processos residentes em cada nodo operador e das informações a respeito das interligações dos portos que estabelecem a rede lógica de comunicação entre os processos.

### VI.4.2.4 PRIMITIVAS DO MECANISMO PARA COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS BASEADO EM PORTOS

São delineadas a seguir as primitivas para a manipulação de portos e as primitivas para comunicação e sincronização entre processos baseado nos portos definidos no item anterior. Não se preocupou, no momento, em descrever as especificidades de cada uma delas, como também, em determinar que esse conjunto de primitivas seja suficiente para todos os tipos de aplicações possíveis. Lembrem-se que o suporte objetiva servir para a construção de Sistemas Operacionais Distribuídos para uso geral. Assim, as necessidades para sistemas de controle de processos ( tempo real ), não foram analisadas, por exemplo. Objetivou-se apenas resumir as funções de cada primitiva para servir como referência para os itens que seguem, onde serão descritos os

comportamentos para fins de recuperação e confiabilidade, para depois, no capítulo VI, especificar cada uma delas com maiores detalhes. Entretanto, com a intenção de complementar as possíveis formas de comunicação ilustradas pelas figuras do item anterior ( tipos de interligações de portos ), resolveu-se delinear as primitivas e suas funções, de acordo com elas, iniciando-se pelas básicas para manipulação de portos locais; ou seja, aquelas para criação, destruição, conexão e desconexão de portos.

## A) PRIMITIVAS PARA MANIPULAÇÃO DE PORTOS

As primitivas necessárias para a manipulação de portos são aquelas citadas em V.1.3, ou sejam:

- para criar porto;
- para destruir porto;
- para conectar portos; e
- para desconectar portos.

As primitivas para criar e destruir portos são:

**Criar_Porto:** Primitiva para criar a estrutura de dados que representa o porto e associá-la a um processo que será o seu dono.

**Eliminar_Porto:** Primitiva para eliminar um porto liberando o espaço de memória por ele ocupado. Essa primitiva pode ser usada apenas pelo processo dono do porto.

As primitivas para conexão e desconexão de portos devem ser usadas apenas pelos processos que têm a função de gerenciar a configuração do sistema ( aplicação ). Simplesmente porque as suas funções são para o estabelecimento da rede lógica de comunicação e de sua alteração dinâmica ou não. Além disso, para que um processo possa criar seus portos locais e também possa estabelecer a conexão com outros não locais, ele precisaria conhecer estes, o que invalidaria o conceito de portos locais. A menos que um processo queira se comunicar consigo, o que é, no mínimo, muito estranho.

**Conectar_Porto:** Primitiva para conectar um porto de saída a um porto de entrada. As possibilidades são aquelas definidas no item VI.4.2.3, ilustradas pelas figuras VI.14 e 15.

**Desconectar_Porto:** Primitiva para desconectar um porto de saída de um porto de entrada.

## B) PRIMITIVAS PARA COMUNICAÇÃO E SINCRONIZAÇÃO

São descritas a seguir as primitivas e suas funções para comunicação e sincronização entre processos baseado em portos. Para maior clareza em termos de utilização das primitivas pelos processos emissor e receptor para o estabelecimento da forma de comunicação e sincronização desejada, se no esquema de "rendez-vous" estendido ou simples ou na forma assíncrona, um para um ou por difusão, segue a descrição com essa intenção.

### ESQUEMA DE "RENDEZ-VOUS" ESTENDIDO

As três primitivas que seguem, devem ser usadas da seguinte forma para o estabelecimento de uma comunicação no esquema de "rendez-vous" estendido: o processo emissor usa a primitiva de envio, enquanto que o processo receptor deve conter as primitivas para recepção da mensagem e para o envio da mensagem resposta, nessa ordem.

**Enviar_PSDRA:** primitiva para enviar uma mensagem a um processo via um porto do tipo PSDRA de forma bloqueada a espera de uma resposta. Tal porto deve estar conectado a um porto do tipo PEDRA;

**Receber_PEDRA:** primitiva para receber mensagem de forma bloqueada de um porto do tipo PEDRA, que deve estar conectado a um porto do tipo PSDRA. Após a recepção da mensagem ela devolve o controle de execução ao processo que solicitou o seu serviço.

**Responde_PEDRA:** primitiva para enviar uma mensagem resposta, de forma bloqueada, à recebida pela primitiva Receber_PEDRA correspondente, cujo destino é para o porto tipo PSDRA, pertencente ao processo emissor da mensagem e que está bloqueado à sua espera.

## ESQUEMA DE "RENDEZ-VOUS" SIMPLES

As duas primitivas que seguem permitem que um par de processos, um emissor e um receptor, se comuniquem de forma síncrona no esquema de "rendez-vous" simples. O processo emissor que envia uma mensagem fica suspenso a espera de um sinal de confirmação de recepção.

**Enviar_PSDCA:** primitiva para enviar uma mensagem a um processo via um porto do tipo PSDCA de forma bloqueada a espera do sinal de confirmação da recepção da mensagem. Tal porto deve estar conectado a um do tipo PEDCA;

**Receber_PEDCA:** primitiva para receber mensagem, de forma bloqueada, de um porto do tipo PEDCA, que logo após, deve enviar o sinal de confirmação de recepção para o porto com o qual este está conectado, que deve ser do tipo PSDCA.

## ESQUEMAS DE COMUNICAÇÃO ASSÍNCRONA

Uma comunicação assíncrona foi definida como sendo estabelecida apenas pelo par de primitivas: enviar não bloqueante e receber bloqueante. Diante do exposto no item anterior, se está interessado em duas formas de comunicação assíncrona, no esquema um para um e um para vários ( difusão ), no caso, mais especificamente, um para um grupo de processos pré-determinado em tempo de configuração do sistema.

**Enviar_PSS:** primitiva para enviar uma mensagem, de forma não bloqueante, a um porto tipo PSS . Tal porto deve estar conectado a um porto do tipo PES;

**Receber_PES:** primitiva para receber mensagem de forma bloqueante de um porto do tipo PES, o qual deve estar conectado a um do tipo PSS. Após a recepção da mensagem, ela devolve o controle de execução ao processo que a solicitou.

**Enviar_PD:** primitiva para enviar uma mensagem na forma de difusão. Para isso, a mensagem é enviada de forma não bloqueante

a um porto do tipo PD. Tal porto deve estar conectado a pelo menos um do tipo PR.

Receber_PR: primitiva para receber mensagem de forma bloqueante de um porto do tipo PR, que deve estar conectado a um porto do tipo PD. Após a recepção da mensagem, ela devolve o controle de execução ao processo que a solicitou.

### VI.4.2.5 CARACTERÍSTICAS DE CONFIABILIDADE

Tratar-se-ão a seguir as questões referentes à confiabilidade levantadas e delineadas neste capítulo até então, que são:

(1) suporte para o mecanismo para recuperação e migração de processos;

(2) suporte para detecção e eliminação de mensagens órfãs; e

(3) suporte para detecção e eliminação de computações órfãs.

### A) SUPORTE PARA O MECANISMO PARA RECUPERAÇÃO E MIGRAÇÃO DE PROCESSOS

Como visto no item VI.4.1, o mecanismo para recuperação e migração de processos, necessita como suporte que toda mensagem transmitida de um processo para um outro seja enviada para ele também.

As funções desse suporte devem ser as seguintes, quando a transmissão da mensagem é local; processos emissor e receptor residem em um mesmo nodo operador:

- enviar primeiro a mensagem para o GRMP e esperar a confirmação de recepção;
- se confirmada, então, colocar a mensagem no porto de entrada destino ( porto de processo destino );
- caso contrário, levantar tal exceção;

Quando a transmissão da mensagem é remota ( o processo destino é remoto ), as funções do suporte devem ser as seguintes:

- enviar primeiro a mensagem para o GRMP e esperar a confirmação da recepção;

- se confirmada, então, enviar a mensagem para o porto de entrada destino ( porto de processo destino remoto ) e esperar confirmação;
- caso confirmada, então, o processamento segue seu caminho normal;
- caso não chegue a confirmação de recepção por parte do GRMP, deve ser levantada uma exceção e, conseqüentemente, não sendo enviada a mensagem para o porto de entrada destino.
- caso não chegue a confirmação de recepção por parte do porto de entrada do processo destino, então, deve ser enviada uma mensagem para o GRMP descartar a mensagem salva para aquele processo cópia e terminar a operação levantando uma exceção.

Note que, dessa forma, os comportamentos do suporte são compatíveis tanto para comunicação síncrona como assíncrona, bastando que, para comunicação assíncrona, a exceção levantada seja vista de forma adequada. Isto é, a exceção levantada pode ser simplesmente ignorada, refletindo a não certeza da recepção da mensagem pelo processo destino, tanto se a transmissão é local ou não.

Diante da necessidade do suporte saber se a mensagem enviada é para um processo remoto ou não, fica determinada a sua localização dentro do contexto do núcleo do sistema. No caso, no "software" de portos ( figura VI.12 ).

Cabe observar que as mensagens de confirmação emitidas pela primitiva Receber_PEDCA, não devem ser enviadas para o GRMP, pois refletem apenas sinais de recepção. Elas têm a função de apenas acordar os respectivos processos emissores das mensagens e porisso, não serão retidos em quaisquer "buffers" para fins de recepção.

## B) SUPORTE PARA DETECÇÃO E ELIMINAÇÃO DE MENSAGENS ÓRFÃS

Como sabido, a retransmissão da mensagem por parte das primitivas para comunicação entre processos, é a única técnica para competir com falhas transientes do subsistema de comunicação, como perda da mensagem. Como conseqüência, ela pode dar surgimentos a eventos não desejáveis denominados de órfãos, descrito no item IV.5 onde também está sugerido um mecanismo para evitá-los, no caso, o M1 ( enumeração seqüencial das mensagens ), para tratar mensagens órfãs, o qual foi admitido sua existência no item VI.3.

Assim, aqui descreve-se apenas uma forma de implementação, que é a seguinte:

(1) quanto às estruturas de dados:

- o núcleo de cada nodo operador mantém o número de seqüência que deve ser associado à próxima mensagem a ser transmitida por algum dos processos sob sua gerência. O valor inicial do número de seqüência será zero, definido quando da iniciação do sistema;
- cada porto receptor mantém uma tabela com os números de seqüências das últimas mensagens recebidas. Um número para cada conexão;

(2) quanto ao esquema para emissão de mensagens:

- cada primitiva de envio de mensagem, primeiro salva o valor do número de seqüência mantido pelo núcleo local, para sua utilização. Depois, atualiza aquele valor mantido pelo núcleo, incrementando-o de um. Em seguida, associa o valor de número de seqüência salvo à mensagem a ser emitida e retransmitida ( se houver a necessidade de ) pelo número máximo de tentativas que deve ser definido como parâmetro nessa primitiva;

(3) quanto ao esquema para detecção e eliminação de mensagens órfãs:

- cada primitiva de recepção, ao receber uma mensagem, antes de colocá-lo no respectivo porto de entrada especificado como um de seus parâmetros, verifica se o número de seqüência da mensagem é maior que aquele constante na tabela na respectiva conexão. Se for, então aquele valor da tabela é atualizado para ser igual ao da mensagem recebida e a mensagem é colocada no porto destino. Caso contrário, a mensagem é simplesmente descartada.

Para o presente trabalho, admitiu--se que os valores do tipo inteiro gerados na forma seqüencial, nunca ultrapassam o valor limite dos inteiros da linguagem utilizada para programação.

Percebe-se que suporte ficaria mais adequadamente colocado a um "software" de troca de mensagens a nível de subsistema de comunicação, o qual, no presente trabalho, será admitido consistindo de apenas duas primitivas, enviar_msg e receber_msg, que serão descritas logo adiante. Entretanto, resolveu-se atribuí-lo ao "software" de portos para deixar o núcleo mais livre possível de informações que devem ser salvas para fins de recuperação de processos e clara a necessidade de

que os números de seqüências devem ser resistentes contra colapso de nodos operadores; o número mantido pelo núcleo e os das tabelas mantidos pelos portos de recepção.

## C) SUPORTE PARA DETECÇÃO E ELIMINAÇÃO DE COMPUTAÇÕES ÓRFÃS

No item VI.3.1 discutiram-se os mecanismos para tratamento de órfãos, ou melhor de computações órfãs, que correspondem às seqüências de operações realizadas decorrentes das comunicações interrompidas, pelos motivos lá descritos; colapso de nodo operador ou de comunicação e estouro do tempo limite definido na primitiva de comunicação. Mais precisamente, somente para as comunicações segundo o esquema de "rendez-vous" estendido, onde a extensão pode-se tornar órfã, segundo as justificativas ali explicitadas.

Naquele item, observa-se que se admitiu implicitamente que o valor do tempo limite era suficiente para uma comunicação síncrona ( comunicação no esquema de "rendez-vous" simples - estabelecida via portos do tipo PSDCA e PEDCA ). Isso permitiu descartar a possibilidade de surgimento de computação órfã para essa forma de comunicação, uma vez que, ela é feita entre duas entidades ativas ( os processos ) e o esquema de recuperação de processos admite que as trocas de mensagens já efetivadas não são afetadas por ele ( os processos devem ser determinísticos ). Porém, no fim deste item discutir-se-á um esquema.

Definiu-se também naquele item os possíveis modos de cooperação entre processos, por meio do esquema de "rendez-vous" estendido, cujos órfãos que porventura possam surgir, podem ser devidamente detectados e eliminados.

A técnica basicamente consiste da utilização do tempo limite e do correspondente "deadline" ( tempo máximo para a execução da extensão do "rendez-vous" ). Detalhando um pouco mais, são as seguintes as tarefas a serem realizadas:

(1) Das primitiva Receber_PEDRA:

- marcar um ponto de recuperação, enviando uma mensagem para o GRMP para estabelecer um ponto de recuperação, cujas informações necessárias para tal estão contidas na própria mensagem e sinalizá-lo de forma a evitar que o estabelecimento

automático não seja feito enquanto o processo estiver executando a extensão do "rendez-vous";

- executar efetivamente as funções normais;
- caso o pedido para marcar um ponto de recuperação não seja bem sucedido, então, passar para a primitiva Responde_PEDRA essa exceção e continuar com as suas tarefas normais;

(2) Das Responde_PEDRA:

- se expirou o "deadline" ou o processo que enviou a mensagem entrou em colapso, percebido pelo estouro do tempo limite dessa primitiva, então:
  (a) solicitar ao GRMP o descarte da mensagem salva na cópia do processo, como também, decrementar o número de mensagens enviadas e recebidas da cópia do processo fonte;
  (b) verificar se o ponto de recuperação foi devidamente estabelecido. Caso sim, então, solicitar ao GRMP a recuperação do processo; Caso não, então, levantar tal exceção, deixando para o processo servidor a tarefa de tratar esse problema.
- caso contrário, executar as suas funções normais;
- sinalizar o GRMP que ele pode, agora, continuar com o seu estabelecimento automático de pontos de recuperação.

(3) Observação: a probabilidade, de que o pedido de marcação de um ponto de recuperação solicitado pela Receber_PEDRA falhar, deve ser muito pequena. Entretanto, se acontecer, provavelmente fato análogo já deverá ter ocorrido e percebido por qualquer uma das primitivas para emissão de mensagens, uma vez que elas também enviam mensagens para o GRMP. Além disso, é considerado que o tempo estimado para isso pode muito bem ser melhor determinado, pois são para primitivas do próprio sistema suporte para as aplicações;

Pela observação feita, nota-se que é interessante que o tempo limite, estimado pelo usuário e definido como parâmetro das primitivas bloqueantes para envio de mensagens a portos, seja no mínimo igual ao pré-definido pelas primitivas de troca de mensagens de mais baixo nível; caso não seja, levantar tal exceção. Seria interessante criar-se uma primitiva que fornecesse esse valor.

Em não fazendo isso, existe a probabilidade de que o tempo limite tenha sido muito mal estimado ( menor que o mínimo necessário ).

Assim, pode ocorrer o estouro do "deadline" durante a execução da parte de recepção de mensagem de uma primitiva do "software" de portos. Conseqüentemente, é importante que ela verifique essa possibilidade e se acontecer, solicitar as tarefas delegadas ao Responder_PEDRA.

Já para o caso de ocorrência de colapso do nodo operador onde reside o processo fonte da mensagem, um esquema objetivando melhorar o desempenho do sistema, é usar um semelhante àquele definido no Rajdoot, isto é, utilizar-se de um contador de colapso ( o "crash-counter" ). Enquanto que naquela técnica o fato de que o processo emissor entrou em colapso e foi recuperado, é detectado somente quando da execução de uma nova chamada ou pelo processo exterminador, nesta impõe-se que as primitivas Receber_PEDRA e Responder_PEDRA, executem as tarefas de detecção e eliminação. A detecção consiste em verificar se o "crash-counter" foi incrementado de um; se foi, executar as tarefas descritas em (2) logo acima, caso contrário, prosseguir normalmente com as suas funções. Para melhorar mais ainda, considerando que a difusão do restabelecimento de um nodo operador ( divulgação de seu novo "crash-counter" a todos os nodos operadores ) é atendida como uma mensagem urgente, fazer com que seja verificado se o processo em execução deve ou não ser interrompido e recuperado como em (2).

Uma outra questão que merece atenção é a seguinte: considerando que o mecanismo de recuperação somente descarta as mensagens já recebidas quando da marcação de um novo ponto de recuperação, seria interessante que a Responde_PEDRA, solicitasse tal pedido. Isso evitaria que caso ocorresse a necessidade da recuperação, com certeza a última comunicação desse tipo não precisaria ser refeita.

Naturalmente, existem outras pequenas questões como essas delineadas. Entretanto, julgando que as primeiras hipóteses admitidas são bastante razoáveis e que, de um modo geral, o trecho da extensão de um "rendez-vous" é relativamente bastante pequeno, muitas das vezes se limitando a apenas verificar a consistência da mensagem recebida, a probabilidade de que um nodo que entrou em colapso e foi restabelecido dentro do intervalo de tempo em que aquela mensagem está em consideração, é praticamente zero. Por esses motivos é que resolveu-se implementar o suporte em questão da forma descrita nos passos (1) e (2), ficando para pesquisas futuras as suposições feitas.

### VI.4.2.6 "SOFTWARE DE REDE

A camada de rede foi definida para criar uma máquina virtual para o "software" de portos, escondendo deste os detalhes do subsistema de comunicação sobre o qual ele realmente se apóia. Para tanto, essa camada oferece àquele "software" duas primitivas por meio das quais ele pode solicitar o envio de mensagens para processos remotos, como também receber mensagens remotas. Dessa forma, o presente trabalho pode-se eximir dos detalhes desse sem prejuízo aos seus objetivos.

Assim sendo, descrevem-se as funções de cada uma das duas primitivas citadas, já referida no item anterior como "software" de troca de mensagens.

- Enviar_Msg ( Nodo_fonte, Proc_Fonte, Porto_Fonte, Nodo_Destino, Proc_Destino, Porto_Destino, End_Buffer, NSM, Tempo_Limite, Status )

  Primitiva utilizada pelas primitivas do "software" de portos, para transmitir uma mensagem a um porto remoto. O algoritmo esquemático da figura VI.16 ilustra o funcionamento dessa primitiva. Os parâmetros têm os seguintes significados:

  - Nodo_Fonte: identificador do nodo operador fonte da mensagem;
  - Proc_Fonte: identificador do processo dono de Porto_Fonte;
  - Porto_Fonte: identificador do porto emissor da mensagem;
  - Nodo_Destino: identificador do nodo operador destino da mensagem;
  - Proc_Destino: identificador do processo dono de Porto_Destino;
  - Porto_Destino: identificador do porto para onde a mensagem deverá ser colocada;
  - End_Buffer: endereço do "buffer" cujo conteúdo é a mensagem a ser enviada;
  - NSM: número de seqüência da mensagem a ser enviada;
  - Tempo_Limite: tempo máximo de espera para que a operação se complete com sucesso.

- Status: informação de retorno indicando se houve ou não sucesso na operação da transmissão da mensagem. Isto é, Status = sucesso ou Status = insucesso ( estouro do tempo limite ).

- Receber_Msg ( Nodo_Fonte, Proc_Fonte, Porto_Fonte, End_Buffer, Tempo_Limite, Status );

Primitiva utilizada pelas primitivas do "software" de portos para receber mensagens de um porto remoto. O algoritmo esquemático da figura VI.17 ilustra o funcionamento dessa primitiva. Os parâmetros têm os seguintes significados:

- Nodo_Fonte: identificador do nodo origem da mensagem. Caso a mensagem pode ser proveniente de qualquer um, esse parâmetro deve conter o valor "ANY";
- Porto_Fonte: identificador do porto emissor da mensagem;
- End_Buffer: endereço do "buffer" para onde a mensagem recebida deve ser colocada;
- Tempo_Limite: tempo máximo de espera para que a operação se complete com sucesso.
- Status: informação de retorno indicando se houve ou não sucesso na operação da transmissão da mensagem. Isto é, Status = sucesso ou Status = insucesso ( estouro do tempo limite ).

Essas duas primitivas por sua vez, se utilizarão de um programa denominado de "servidor de rede", cujas funções principais são:

- enviar uma mensagem para o nodo operador destino especificado; e
- receber e avisar o núcleo da chegada de uma mensagem se esta não estiver corrompida;

Isso permite que o núcleo tenha condição de verificar se uma mensagem é para ele executar algumas de suas tarefas, como por exemplo, salvar o estado de um processo. A figura VI.18 mostra a visão funcional do "software" de troca de mensagens.

```
alocar um "buffer" de transmissão;
colocar o "buffer" do porto no "buffer" de transmissão;
fazer Status do "buffer" de transmissão = "não
                                          transmitido";
suspender o processo;
desalocar o "buffer" de transmissão;
se o processo foi acordado devido ao estouro do
    tempo_limite
     então
            fazer Status do Enviar_Msg = "tempo limite estourado";
            retornar o controle de execução ao chamador dessa
            primitiva;
     senão
            fazer Status do Enviar_Msg = "sucesso"
```

Figura VI.16: Algoritmo da primitiva Enviar_Msg

```
se não há mensagem no "buffer" de recepção para o porto
    receptor
     então
            suspender o processo;
se o processo foi acordado devido ao estouro do tempo
    limite
     então
            fazer Status do Receber_Msg = "tempo limite estourado";
            retornar o controle de execução ao chamador dessa
            primitiva;
colocar o "buffer" de recepção no "buffer" do porto;
desalocar o "buffer" de recepção;
fazer Status do Receber_Msg = "sucesso";
```

Figura VI.17: Algoritmo da primitiva Receber_Msg

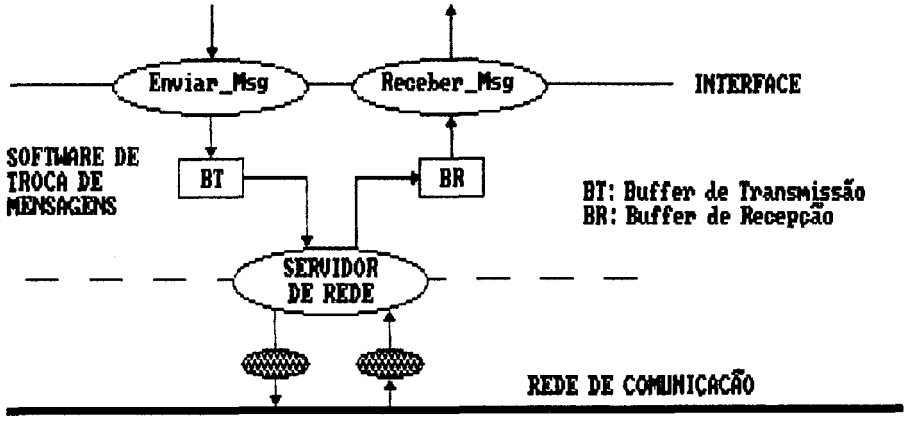



Figura VI.18: Visão funcional do "software" de troca de mensagens

# CAPÍTULO VII

## IMPLEMENTAÇÃO DE UM SUPORTE PARA SISTEMA OPERACIONAL DISTRIBUíDO COM RECONFIGURAÇÃO DINÂMICA DE PROCESSOS

O objetivo deste capítulo é descrever o Projeto de um Suporte para Sistema Operacional Distribuído com Reconfiguração Dinâmica de Processos, em desenvolvimento, conforme os mecanismos descritos no capítulo VI.

Não se desenvolveu toda uma máquina virtual de baixo nível. Preocupou-se apenas com os recursos que tal suporte deve oferecer para se poder construir um sistema de aplicação distribuído tolerante a falhas, provenientes de faltas de "hardware". Problemas relativos à interface núcleo e subsistema de comunicação não foram abordados, salvo em alguns pontos que se julgou necessário. Também, foram admitidos a existência de certos recursos de "hardware" e de "software" de baixo nível, para a implementação das primitivas objetos do presente projeto.

Enfim, preocupou-se em implementar os mecanismos para comunicação e sincronização entre processos por meio de portos locais, com suporte para detecção e eliminação de órfãos, mecanismos para

recuperação de processos e mecanismos para migração de processos, todos desenvolvidos no capítulo VI.

Para tanto, implementou-se um ambiente multitarefas, composto de um escalonador e das primitivas para o gerenciamento de processos e do mecanismo para a configuração inicial do Sistema Operacional Distribuído, ou de forma genérica, Sistema Distribuído. Quanto à reconfiguração dinâmica do sistema, como visto naquele capítulo deverá ser exercida pelos mecanismos para recuperação e migração de processos, quando da detecção de falha de algum nodo operador ( item VI.4.1 ) ou quando do recebimento de pedido para migração de processos feito pelo mecanismo para balanceamento dinâmico de carga ( não abordado ).

A linguagem utilizada para a implementação foi a Modula-2 ( LOGITECH, 1986 ), por ser uma que incorpora muitos dos conceitos de linguagens modernas para programação de sistemas, como esconder informações, tipos abstratos de dados, programação modular, programação estruturada, compilação em separado e principalmente, por conter um núcleo com o mínimo de recursos, como: conceito de processo a nível de co-rotina e facilidades para o acesso a recursos de máquina, e se encontrar disponível para máquinas de pequeno porte como os Computadores Pessoais, tipo IBM-PC XT. A quantidade mínima de recursos de baixo nível oferecida pela linguagem permite que se vá criando à medida do necessário, os de maior nível de acordo com os algoritmos desejados. Construindo dessa forma, o ambiente que mais se adequa às tarefas destinadas. Com isso, não se quer dizer que com uma linguagem que contenha um núcleo com maiores recursos, como ADA, por exemplo, que oferece o conceito de tarefa ( processo ), primitivas para troca de mensagens ( o "rendez-vous" estendido ), primitivas para comunicação não determinística, etc, se fique impedido de construir outros ambientes. Para isso, entretanto, pode ser que muitos desses recursos disponíveis não possam ser utilizados diante de suas políticas não satisfazerem os algoritmos mais apropriados para o sistema a ser implementado. Por exemplo, a comunicação e sincronização entre processos contida na linguagem ADA é o "rendez-vous" estendido, porém o que deseja neste trabalho é que tal mecanismo seja aquele baseado em portos, que é um de nível um pouco mais baixo. Em contrapartida, ADA apresenta, por exemplo, como construção de linguagem o comando para comunicação não determinística que se for

construída em Modula-2, essa facilidade fica com a aparência um tanto forçada.

Assim, a seguir descrevem-se as hipóteses sobre as quais foi desenvolvido o projeto, a estruturação do núcleo ( o ambiente multitarefas suporte ), suas primitivas e os mecanismos objetos da tese, finalizando com a descrição de um programa configurador inicial do sistema.

## VII.1 OBJETIVOS E HIPÓTESES

Dentre as várias funções que um núcleo para sistemas do tipo e porte em questão, deve oferecer, preocupou-se com aquelas direcionadas para dar suporte para migração de processos, para recuperação de processos e para o sistema de troca de mensagens tolerante a falhas, com detecção e eliminação de órfãos, conforme descritos em VI. Ficaram para posteriores pesquisas, mecanismos para detecção de "deadlock", suporte para o sistema de arquivo, suporte para "spooling" de entrada e de saída, gerenciamento de processos de usuários, mecanismos para balanceamento dinâmico de cargas, etc.

Os mecanismos a serem descritos são, então:

1) mecanismos para comunicação e sincronização baseadas em portos tipificados com detecção e eliminação de órfãos; e
2) mecanismos para recuperação de processos, para tratar falhas provenientes dos seguintes tipos de faltas de "hardware":
   - colapso de nodos operadores ou os seus isolamentos por defeito de seus nodos de comunicação;
   - falhas transientes da rede de comunicação;
3) mecanismo para a migração de processos, visando:
   - oferecer facilidades para o balanceamento dinâmico de cargas;
   - oferecer facilidades para o crescimento incremental ou o decrescimento voluntário ( desativação voluntária de algum nodo operador - conforme citada no item IV.4 ) e, assim, permitindo que o sistema seja reconfigurável dinamicamente; e
   - defeito de dispositivos periféricos;

Como visto existem vários problemas para a implementação desses tipos de mecanismos, dependendo da infra-estrutura de "hardware". Assim, fizeram-se a seguir, as hipóteses sobre as quais se basearam as implementações, que são:

(a) rede local baseada em barramento confiável, no sentido de que existam linhas redundantes com comutação automática;
(b) existência de um protocolo simples para a comunicação entre nodos operadores - o protocolo apenas detectará falha na comunicação, mas não tratará, apenas reportará aos seus usuários na forma de exceção;
(c) os nodos operadores são monoprocessadores e idênticos;
(d) a existência de um nodo especial superprotegido que terá funções específicas como: iniciar o sistema, reconfigurar o sistema e recuperar processos;
(e) existência de dispositivos periféricos compartilhados e seus respectivos monitores ( "drivers" );
(f) existência de dispositivos para entrada de tarefas de usuários e respectivos monitores;

Dentre os dois modelos de construção de Sistemas Operacionais Distribuídos definiu-se pelo simétrico, por este englobar o outro; o modelo usuário-servidor é uma particularização daquele. Dessa forma, para fins de maior abrangência, se admitiu:

- existência de redundância ativa dos dispositivos compartilhados "críticos", tais como impressora e disco magnético de alta capacidade ( a redundância é ativa no sentido de que ela pode ser utilizada particularmente pelos usuários locais );

Além dessas hipóteses existem outras que já foram admitidas para o desenvolvimento do presente trabalho, que foram:

- a superproteção de um nodo é para garantir que todas as informações importantes para fins de recuperação e migração involuntária de processos nunca se perderão, o seu colapso implicará no colapso do sistema. Assim, a princípio, não haverá a necessidade de se criar cópias dos processos residentes nesse nodo, principalmente, porque eles têm as funções de gerenciar o sistema, para fins delineados em (d);
- todos os processos do sistema deverão estar corretamente especificados e implementados. Não deverá, então, existir, como também, surgir falhas de "software" como acenadas por HORNING et alii ( 1974 ), por exemplo. Assim, a presença de defeito em algum componente de um nodo operador deve corresponder à sua parada instantânea de funcionamento. Isso para garantir que os problemas que possam advir, não provoquem qualquer efeito colateral em seus processos, fazendo com que eles saiam de suas especificações. Para

tal, pode se usar a estruturação proposta por RANDELL ( 1984 ), ou seja, a da utilização de componentes idealizados tolerantes a falhas.

Apesar de a hipótese (d), mais detalhada logo em seguida, e do suporte para recuperação de processos, sugerirem a construção de dois tipos de núcleos, um para ser replicado em cada nodo operador comum e um para o nodo especial, optou-se pela construção de apenas um, pelos seguintes motivos, em ordem:

- o núcleo do nodo especial deverá, praticamente, oferecer o mesmo ambiente que os para os nodos operadores comuns, uma vez que as diferenças são pequenas particularidades como, a não necessidade de gerenciar processos de usuários e do suporte para recuperação de processos e o comportamento do seu "bootstrap loader" primário ( ou "cold start" - partida fria ), como serão melhores observadas quando da descrição das funções do núcleo;
- o nodo operador superprotegido pode ser definido como sendo qualquer um do Sistema Distribuído; e
- a proteção pode ser reforçada por meio do uso de redundância passiva ou ativa.

Quanto ao suporte para a tolerância a falhas de "software", não se entrará em maiores detalhes além daqueles descritos nos capítulos anteriores. Limitou-se, no fim deste, a apenas sugerir a utilização de alguns mecanismos, mais para se ter uma visão global de seus usos, como a aventada na hipótese referente ao comportamento dos processos.

## VII.2 NÚCLEO

Seguindo o capítulo V, item V.1, o núcleo foi estruturado em duas camadas hierárquicas. A camada inferior, se constituindo na primeira máquina virtual, com a função de escondendo os detalhes do "hardware" subjacente da camada superior, a qual é feita oferecendo todas as operações ( primitivas Enviar_Msg e Receber_Msg ) para a comunicação entre nodos operadores do Sistema Distribuído, que são as necessárias para o presente trabalho ( veja capítulo VI ). A camada superior deverá, por sua vez, criar uma máquina virtual de nível mais alto, que no caso, se restringe ao oferecimento das funções de:

(a) oferecer um ambiente multitarefas e primitivas para o gerenciamento dos processos locais;

(b) esconder dos processos sob sua gerência as localizações dos processos parceiros remotos, permitindo que eles sejam codificados sem qualquer preocupação a respeito da infra-estrutura existente que cria essa visão;

(c) oferecer primitivas para a comunicação e sincronização entre processos baseado no conceito de portos locais, com suporte para detecção e eliminação de órfãos;

(d) oferecer suporte para os mecanismos para recuperação e migração de processos; e

(e) oferecer primitivas para configuração inicial do sistema.

Outras primitivas, como para o gerenciamento de memória, sistema de arquivo, etc, não serão descritas. Será admitida a existência dos próprios recursos completos; isto é, a existência de um gerenciador de memória, um sistema de arquivo, etc., cada um estruturado de forma confiável e segura, explicitando as falhas não possíveis de serem por eles devidamente tratadas, para o seu ambiente. De qualquer forma, à medida do necessário para maior clareza, se fará observações a respeito.

O núcleo para poder exercer a função (b), deve conter informações referentes às localizações de todos os processos do sistema, de modo que as primitivas de comunicação e sincronização entre processos baseado em portos, oferecidas como interface dessa camada para as superiores, as usem para o estabelecimento efetivo da comunicação. Tais informações, neste trabalho, se resumiram em tabelas estruturadas, onde ficam explicitados os dados necessários para a execução da tarefa do estabelecimento de uma comunicação solicitada

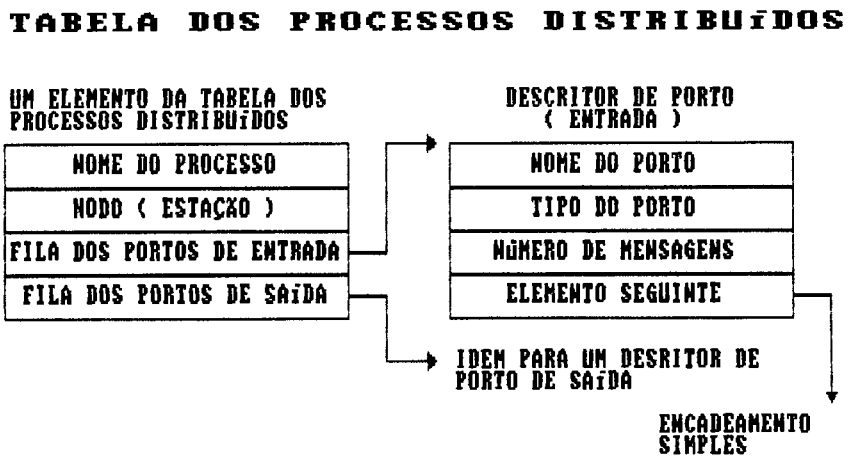



Figura VII.1 Estrutura de dados para determinação da localização de um processo

por algum processo. A figura VII.1 ilustra essas tabelas. Como pode ser notado, elas foram elaboradas para serem independentes do nodo em que elas residirão.

As outras funções (a), (c), (d) e (e), serão descritas a seguir.

## VII.2.1 DESENVOLVIMENTO DO AMBIENTE MULTITAREFAS E DAS PRIMITIVAS PARA GERENCIAMENTO DE PROCESSOS

Em um ambiente multitarefas, os processos em execução concorrente podem estar em um de cinco estados possíveis, a saber ( veja figura VII.2 ):

- **novo**: processo a ser introduzido ao ambiente;
- **pronto para execução**: processo criado esperando por processador para ser executado;
- **executando**: processo em execução ( de posse do processador );
- **suspenso** ( ou **bloqueado** ): processo que deseja ou necessita esperar que alguma condição seja satisfeita, para poder seguir com o processamento normal;
- **terminado**: processo que quer ou deve sair do ambiente;

Nota-se que a transição de um processo de um estado para um outro não deve ocorrer de forma aleatória, mas deve seguir alguma lógica, por exemplo: quando um processo está no **estado de novo**, ele só pode ir para o **estado de pronto para execução**; depois desse estado, ele pode ir para o **estado de executando**; a partir desse pode ser passado para o **estado dos suspensos** ou para o **estado dos terminados** ou, novamente, para o **estado dos prontos para execução**; caso ele esteja no **estado dos suspensos**, ele pode ir para o **estado de executando**, se a condição pela qual ele estava esperando foi satisfeita, ou para o **estado de pronto para execução**; caso ele esteja no **estado de terminados** nada mais há o que fazer, pois ele deve ser retirado do ambiente. Conseqüentemente, deve-se definir operações por meio das quais se possa exercer tal tarefa ( transição ) quando necessário. No contexto de Sistemas Operacionais, é comum denominar essas operações como primitivas para o gerenciamento de processos.

Para o presente trabalho, o ambiente multitarefas criado é composto de um escalonador de curto prazo, estruturas de dados e das primitivas para o gerenciamento dos processos.

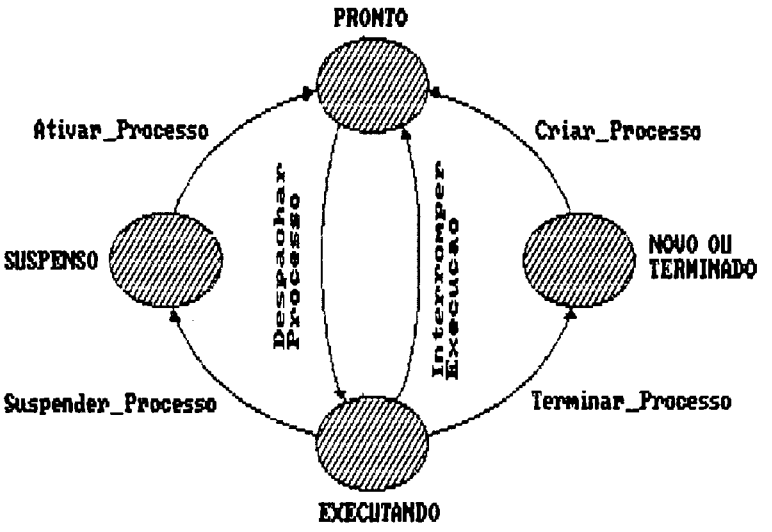



Figura VII.2: Estados dos Processos

O escalonador de curto prazo, segundo as hipóteses estabelecidas, simulará a existência de mais do que um processador, fazendo com que o único processador existente, seja compartilhado pelos processos, da seguinte forma: a cada intervalo de tempo bem determinado, sinalizado pelo relógio do processador por meio de interrupção, ele atribui um novo processo ao processador, salvando antes o que estava em execução de modo a poder, posteriormente, dar continuidade à sua execução, como se ele não tivesse cedido por alguns instantes o processador a um outro ( HOLT et alii, 1983, por exemplo ).

As estruturas de dados principais para a concretização dos possíveis estados em que um processo pode estar, conforme ilustra a figura VII.2, são:

- o descritor de processos: estrutura de dados para reter de forma organizada as seguintes informações a respeito de um processo, por exemplo:
  . o identificador do processo;
  . o estado do processo;
  . prioridade;
  . condição de suspensão;
  . o tempo que falta para terminar a sua fatia de tempo;
  . etc.

Enfim, informações a respeito de um processo para permitir o seu gerenciamento de acordo com as políticas de escalonamento ( no caso, "round-robin" ), condições de suspensão, etc.

- fila para colocação de descritores dos processos prontos para a execução ( representando o estado dos processos prontos para execução );
- fila dos processos em execução ( representando o estado dos processos em execução, no caso apenas um );
- fila dos processos terminados ( representando o estado dos processos terminados ); e
- filas dos processos suspensos por condições ( representando o estado dos processos bloqueados ).

As operações para a manipulação dessas estruturas de dados são, essencialmente:

- criar um descritor de processo;
- eliminar um descritor de processo;
- retirar um descritor de processo de uma fila; e
- colocar um descritor de processo em uma fila;

as quais são utilizadas pelas seguintes primitivas básicas necessárias para o gerenciamento de processos:

- **criar processo**: para criar um descritor de processo, preencher com os devidos dados e colocá-lo na fila dos processos prontos para execução;
- **despachar processo**: para escalonar um processo da fila dos prontos para execução e colocá-lo efetivamente em execução, o qual é feito mapeando na máquina o estado do processo contido no descritor;
- **interromper a execução**: para retirar involuntariamente o processo em execução, salvar o seu estado no respectivo descritor de processo e colocá-lo na fila dos prontos para execução;
- **suspender processo**: para retirar voluntariamente o processo em execução, salvar o seu estado no respectivo descritor de processo e colocar tal descritor em uma das filas dos suspensos por condição;
- **ativar processo**: para retirar um descritor de processo de uma das filas dos suspensos por condição e colocá-lo na fila dos prontos para execução;
- **terminar processo**: para eliminar o descritor de processo, a pilha de trabalho associada, o seu código, etc. Enfim, devolver ao gerente de memória o espaço ocupado por esse processo.

Além dessas primitivas básicas, outras foram necessárias para tornar o núcleo mais flexível, como, por exemplo, a primitiva inicia fila de processos, por meio da qual se criará dinamicamente filas de processos, segundo uma estrutura bem definida a qual ficará escondida dos usuários ( conceito de esconder informação ) e, assim, evitando que eles as utilizem de forma indevida. As figuras VII.3 e VII.4, ilustram os esquemas das estruturas de dados.

A seguir descrevem-se as primitivas para comunicação e sincronização entre processos, visando suprir o núcleo com essa facilidade.

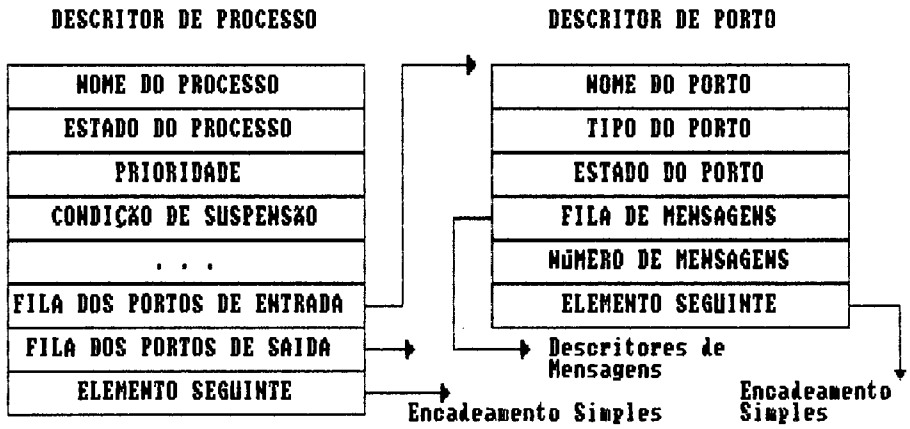



Figura VII.3: Descritor de Processo e de Porto

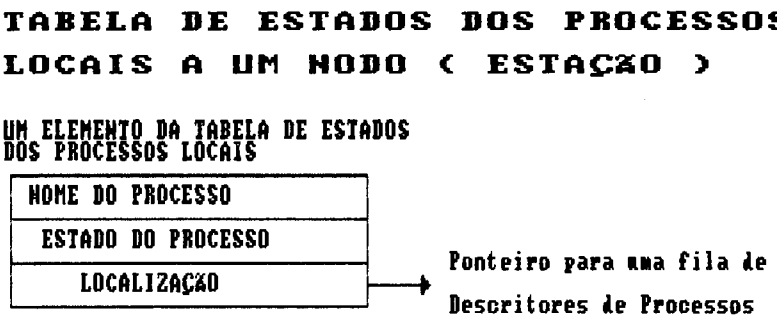



Figura VII.4: Tabela dos Estados dos Processos Locais

## VII.2.2 PRIMITIVAS DE COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS BASEADO EM PORTOS, COM SUPORTE PARA DETECÇÃO E ELIMINAÇÃO DE ÓRFÃOS

Neste item descrevem-se as primitivas para comunicação e sincronização baseado em portos locais que o núcleo deverá oferecer como facilidade para que os processos de um Sistema Operacional Distribuído possam cooperar entre si. As primitivas são aquelas discutidas no item VI.4.2.4. Assim, aqui se limitará a apenas transcrevê-las especificando-as com um pouco mais de detalhes.

### A) PRIMITIVAS PARA MANIPULAÇÃO DE PORTOS

**Criar_Porto ( Iden_Porto, Tipo_Porto, Tipo_Msg, N_Buffers, Status );**

Primitiva para criar a estrutura de dados que representa o porto e associá-la ao processo que a executou que será o seu dono. Cada parâmetro tem o seguinte significado:

**Iden_Porto:** identificador a ser associado ao porto;

**Tipo_porto:** especificação de um dos oito tipos de portos ( PSDRA, PEDRA, PSDCA, PEDCA, PSS, PES, PD ou PR );

**Tipo_Msg:** especificação da capacidade do porto em termos do tipo da mensagem que ele pode aceitar.

**N_Buffers:** especificação do número de "buffers" que deverá ser associado ao porto. Para os portos do tipo PSDRA, PEDRA, PSDCA e PEDCA o valor desse parâmetro deverá ser igual a 1 ( um ), pois a comunicação possível por meio deles é síncrona.

**Status:** informação de retorno que indica se houve ou não sucesso da operação; se não houve qual foi a causa, como segue:

- já existe um porto com esse identificador;

**Eliminar_Porto ( Iden_Porto, Status );**

Primitiva para eliminar um porto liberando o espaço de memória por ele ocupado. Essa primitiva pode ser usada apenas pelo processo dono do porto. Os parâmetros têm os seguintes significados:

**Iden_Porto:** identificador do porto;

**Status:** informação de retorno que indica se houve ou não sucesso da operação; se não houve qual foi a causa, como segue:

- o porto não existe;
- o porto está conectado com algum outro;

**Conectar_Porto ( Porto_Saida, Proc_Emissor, Porto_Entrada, Proc_Receptor, Status );**

Primitiva para conectar um porto de saída a um porto de entrada. As possibilidades são aquelas definidas no item VI.4.2.3, ilustradas pelas figuras VI.14 e 15. Os parâmetros têm os seguintes significados:

**Porto_Saida:** identificador de um porto de saída;

**Proc_Emissor:** identificador do processo dono do porto de saída Porto_Saida;

**Porto_Entrada:** identificador de um porto de entrada;

**Proc_Receptor:** identificador do processo dono do porto de entrada Porto_Entrada;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- inexistência dos processos especificados;
- inexistência dos portos especificados;
- incompatibilidade de interligação: tentativa de conexão entre dois portos de entrada ou entre dois portos de saída;
- incompatibilidade das capacidades dos portos: tipos de mensagens diferentes;

**Desconectar_Porto ( Porto_Saida, Proc_Emissor, Porto_Entrada, Proc_Receptor, Status );**

Primitiva para desconectar um porto de saída de um porto de entrada. Os parâmetros têm os seguintes significados:

**Porto_Saida:** identificador de um porto de saída;

**Proc_Emissor:** identificador do processo dono do porto de saída Porto_Saida;

**Porto_Entrada:** identificador de um porto de entrada;

**Proc_Receptor:** identificador do processo dono do porto de entrada Porto_Entrada;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- inexistência dos processos especificados;
- inexistência dos portos especificados associados aos respectivos processos identificados;

As estruturas de dados para portos estão ilustrados na figura VII.5.

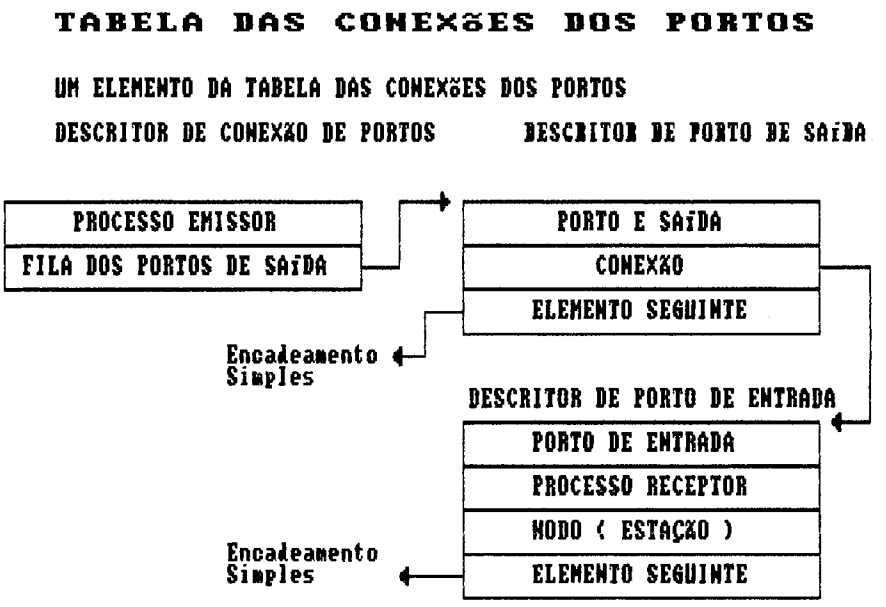



Figura VII.5: Tabela das Conexões dos Portos

## B) PRIMITIVAS PARA COMUNICAÇÃO E SINCRONIZAÇÃO

### ESQUEMA DE "RENDEZ-VOUS" ESTENDIDO

**Enviar_PSDRA ( Porto, Mensagem, Resposta, Tempo_Limite, N_Tentativas, Status );**

primitiva para enviar uma mensagem a um processo via um porto do tipo PSDRA de forma bloqueada a espera de uma resposta. Tal porto deve estar conectado a um porto do tipo PEDRA. Os parâmetros têm os seguintes significados:

**Porto:** identificador de um porto do tipo PSDRA;

**Mensagem:** mensagem a ser enviada para o porto Porto;

**Resposta:** mensagem resposta a ser recebida pelo porto Porto;

**Tempo_Limite:** tempo máximo de espera para que a operação se complete, ou seja, para que a mensagem seja enviada e recebida uma resposta;

**N_Tentativas:** número máximo de tentativas para que a operação se complete com ou sem sucesso. Cada tentativa é feita cada vez que o Tempo_Limite estoura;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- o porto especificado não existe;
- o porto não é do tipo PSDRA;
- incompatibilidade entre a mensagem e a capacidade do porto;
- o porto não está conectado com nenhum outro porto;
- operação encerrada após tentar as N_Tentativas de executá-la;

**Receber_PEDRA ( Porto, Mensagem, Tempo_Limite, Status );**

primitiva para receber mensagem de forma bloqueada de um porto do tipo PEDRA, que deve estar conectado a um porto do tipo PSDRA. Após a recepção da mensagem ela devolve o controle de execução ao processo que solicitou o seu serviço. Os parâmetros têm os seguintes significados:

**Porto:** identificador de um porto do tipo PEDRA;

**Mensagem:** mensagem a ser recebida pelo porto Porto;

**Tempo_Limite:** tempo máximo de espera para que cheque uma mensagem no porto em questão;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- o porto especificado não existe;
- o porto não é do tipo PEDRA;
- incompatibilidade entre a mensagem e a capacidade do porto;
- o porto não está conectado com nenhum outro porto;
- operação encerrada por estouro do tempo limite;

**Responder_PEDRA ( Porto, Mensagem, Tempo_Limite, N_Tentativas, Status );**

primitiva para enviar uma mensagem resposta, de forma bloqueada, à recebida pela primitiva Receber_PEDRA

correspondente, cujo destino é para o porto tipo PSDRA, pertencente ao processo emissor da mensagem e que está bloqueado à sua espera. Os parâmetros têm os seguintes significados:

**Porto:** identificador do porto Porto da respectiva primitiva Receber_PEDRA;

**Mensagem:** mensagem enviada para o porto do tipo PSDRA conectado ao respectivo porto de tipo PEDRA especificado como Porto;

**Tempo_Limite:** tempo máximo de espera para que a operação se complete, ou seja, para que a mensagem seja enviada ao seu destino;

**N_Tentativas:** número máximo de tentativas para que a operação se complete com ou sem sucesso. Cada tentativa é feita cada vez que o Tempo_Limite estoura;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- o porto especificado não existe;
- o porto não é do tipo PEDRA;
- o porto não é aquele especificado pela respectiva primitiva Receber_PEDRA;
- incompatibilidade entre a mensagem e a capacidade do porto;
- operação encerrada após tentar as N_Tentativas de executá-la;

## ESQUEMA DE "RENDEZ-VOUS" SIMPLES

**Enviar_PSDCA ( Porto_Saida, Mensagem, Resposta, Tempo_Limite, N_Tentativas, Status );**

Primitiva para enviar uma mensagem a um processo via um porto do tipo PSDCA de forma bloqueada a espera do sinal de confirmação da recepção da mensagem. Tal porto deve estar conectado a um do tipo PEDCA. Os parâmetros têm os seguintes significados:

**Porto:** identificador de um porto do tipo PSDCA;

**Mensagem:** mensagem a ser enviada para o porto Porto;

**Resposta:** mensagem resposta a ser recebida pelo porto Porto;

**Tempo_Limite:** tempo máximo de espera para que a operação se complete, ou seja, para que a mensagem enviada chegue ao seu destino;

**N_Tentativas:** número máximo de tentativas para que a operação se complete com ou sem sucesso. Cada tentativa é feita cada vez que o Tempo_Limite estoura;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- o porto especificado não existe;
- o porto não é do tipo PSDCA;
- incompatibilidade entre a mensagem e a capacidade do porto;
- o porto não está conectado com nenhum outro porto;
- operação encerrada após tentar as N_Tentativas de executá-la;

**Receber_PEDCA ( Porto_Entrada, Mensagem, Tempo_Limite, Status );**

primitiva para receber mensagem, de forma bloqueada, de um porto do tipo PEDCA, que logo após, deve enviar o sinal de confirmação de recepção para o porto com o qual este está conectado, que deve ser do tipo PSDCA. Os parâmetros têm os seguintes significados:

**Porto:** identificador de um porto do tipo PEDCA;

**Mensagem:** mensagem a ser recebida pelo porto Porto;

**Tempo_Limite:** tempo máximo de espera para que cheque uma mensagem no porto em questão;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- o porto especificado não existe;
- o porto não é do tipo PEDCA;
- incompatibilidade entre a mensagem e a capacidade do porto;
- o porto não está conectado com nenhum outro porto;
- operação encerrada por estouro do tempo limite;

## ESQUEMAS DE COMUNICAÇÃO ASSÍNCRONA

**Enviar_PSS ( Porto, Mensagem, Status );**

primitiva para enviar uma mensagem, de forma não bloqueante, a um porto tipo PSS . Tal porto deve estar conectado a um porto do tipo PES. Os parâmetros têm os seguintes significados:

**Porto:** identificador do porto do tipo PSS para o qual se deseja enviar a mensagem Mensagem;

**Mensagem:** mensagem a ser colocada no porto Porto;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- não existe o porto especificado;
- o porto especificado não é do tipo PSS;
- incompatibilidade entre o tipo da mensagem e a capacidade do porto;
- o porto não está conectado com nenhum outro porto;
- operação encerrada sem sucesso por outros problemas;

**Receber_PES ( Porto, Mensagem, Tempo_Limite, Status );**

primitiva para receber mensagem de forma bloqueante de um porto do tipo PES, o qual deve estar conectado a um do tipo PSS. Após a recepção da mensagem, ela devolve o controle de execução ao processo que a solicitou. Os parâmetros têm os seguintes significados:

**Porto:** identificador do porto do tipo PES por meio do qual se deseja receber uma mensagem;

**Mensagem:** mensagem obtida do porto Porto;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- não existe o porto especificado;
- o porto especificado não é do tipo PES;
- incompatibilidade entre o tipo da mensagem e a capacidade do porto;
- o porto não está conectado com nenhum outro porto;
- operação encerrada sem sucesso por outros problemas;

**Enviar_PD ( Porto, Mensagem, Status );**

primitiva para enviar uma mensagem na forma de difusão. Para isso, a mensagem é enviada de forma não bloqueante a um porto do tipo PD. Tal porto deve estar conectado a pelo menos um do tipo PR. Os parâmetros têm os seguintes significados:

**Porto:** identificador do porto do tipo PD para o qual se deseja enviar a mensagem Mensagem;

**Mensagem:** mensagem a ser colocada no porto Porto;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- não existe o porto especificado;
- o porto especificado não é do tipo PD;
- incompatibilidade entre o tipo da mensagem e a capacidade do porto;
- o porto não está conectado com nenhum outro porto;
- operação encerrada sem sucesso por outros problemas;

**Receber_PR ( Porto, Mensagem, Tempo_Limite, Status );**

primitiva para receber mensagem de forma bloqueante de um porto do tipo PR, que deve estar conectado a um porto do tipo PD. Após a recepção da mensagem, ela devolve o controle de execução ao processo que a solicitou. Os parâmetros têm os seguintes significados:

**Porto:** identificador do porto do tipo PR por meio do qual se deseja receber uma mensagem;

**Mensagem:** mensagem obtida do porto Porto;

**Status:** informação de retorno que indica se houve ou não sucesso da operação. As causas de insucesso pode ser:

- não existe o porto especificado;
- o porto especificado não é do tipo PR;
- incompatibilidade entre o tipo da mensagem e a capacidade do porto;
- o porto não está conectado com nenhum outro porto;
- operação encerrada sem sucesso por outros problemas;

O suporte para detecção e eliminação de órfãos são aquelas descritas em VI.4.2.5. B) e C), que a seguir serão delineadas para fins ilustrativos e de complementação, como também, o suporte para os

mecanismos para recuperação e migração de processos, descritos no item VI.4.2.5.A).

## VII.2.3 SUPORTE PARA OS MECANISMOS PARA RECUPERAÇÃO E MIGRAÇÃO DE PROCESSOS E SUPORTE PARA TRATAMENTO DE ÓRFãOS

No capítulo VI, item VI.4.1, descreveu-se o mecanismo para recuperação e migração de processos que basicamente consiste de um processo denominado de GRMP, que deverá ficar residente no nodo operador especial e no item VI.4.2.5 A), o suporte que o núcleo de cada nodo operador comum deve oferecer para os propósitos do referido mecanismo, no item VI.4.2.5 B), o suporte para detecção e eliminação de mensagens órfãs e no item VI.4.2.5. C), o suporte para detecção e eliminação de computações órfãs.

Assim, o único objetivo deste item é o de descrever com maiores detalhes esse suporte, mostrando como está sendo implementado.

Como visto, o núcleo de cada nodo operador e o núcleo do nodo especial, onde reside o GRMP, interagirão de forma cooperativa para alcançar os objetivos definidos naqueles itens. Nessa cooperação notou-se que as mensagens têm diferentes propósitos: mensagens para serem simplesmente armazenadas e mensagens que na realidade são comandos, ou sejam, são pedidos para execução de serviços. Por exemplo, o GRMP pode receber mensagens para os processos cópias para fins de recuperação desses processos, como mensagens para execução de serviços de recuperação de um processo ou para descartar mensagens das cópias dos processos etc. O núcleo de um nodo operador comum, por sua vez, pode receber mensagens provenientes de processos do sistema, como também mensagens do GRMP, para ele executar a tarefa de recuperação ou migração de um processo etc.

Para isso, criou-se no núcleo um processo, que daqui por diante será referido como processo gerente de recepção, com a tarefa de fazer a distinção dos tipos de mensagens que chegam e ativar as operações adequadas; colocar simplesmente a mensagem no "buffer" do porto receptor destino, ou executar a migração de um processo, etc. Tal processo, naturalmente, deve ser ativado toda vez que chegar uma mensagem remota, serviço da primitiva Receber_Msg do "software" de rede ( ou troca de mensagens, como já assim, referido ). Dessa forma,

toda comunicação entre nodos operadores, independentemente dos objetivos das mensagens, se dará entre processos desse tipo ( o processo gerente de recepção ), que guardadas as devidas proporções vai de encontro com a idéia de comunicação entre grupos de processos ( LIANG et alii, 1990 ). De forma análoga, deveria-se criar um processo gerente de emissão. Não se criaria um único processo desse tipo para enviar e receber mensagens remotas, por questões de clareza e simplicidade; cada um teria suas funções bem específicas ou só para analisar mensagens que chegam ou só para analisar as mensagens que deverão ser enviadas.

Considerando-se que se está utilizando o conceito de portos locais como mecanismo para comunicação e sincronização entre processos, o processo gerente de recepção e o processo gerente de emissão, poderiam criar seus portos de entrada e de saída, respectivamente, que deverão ser conectados aos respetivos processos, também desse tipo do núcleo especial:

- porto de saída de um processo gerente de emissão do núcleo de um nodo operador comum, ao porto de entrada do processo gerente de recepção do núcleo do nodo especial; e
- porto de saída do processo gerente de emissão do núcleo do nodo especial, ao porto de entrada de um processo gerente de recepção do núcleo de um nodo operador comum.

Entretanto, considerando-se que esses processos estão no nível do núcleo, poder-se-ia utilizar de mecanismos de comunicação de mais baixo nível e assim, evitar-se-ia a necessidade de se estabelecer as conexões entre os processos desse tipo, que aliás, são estáticas, pois, um núcleo nunca migrará para um outro nodo. Porém, resolveu-se ficar no meio termo. Isto é, utilizar-se do conceito de portos globais para esses tipo de processos por, justamente, refletir melhor os objetivos dos referidos processos, eles são uma espécie de servidor. Assim, não se precisará estabelecer conexões, como também, não se necessitará do processo gerente de emissão, ficando portanto, apenas como o processo gerente de recepção. Dessa forma, foi necessário criar as primitivas para enviar mensagens para portos globais. Basicamente seria necessário apenas uma, a Enviar_Msg_Porto_Global ( Porto_Global, ... ), por exemplo. Porém, resolveu-se criar mais do que uma, com a intensão de cada uma especificar o tipo de mensagem que ela envia, o que as torna mais confiáveis. Assim sendo, as primitivas são:

- Enviar_Msg_GRMP ( ... );
- Enviar_Serv_GRMP ( ... );
- Enviar_Msg_Núcleo ( Ident_Núcleo, ... ); e
- Enviar_Serv_Nucleo ( Ident_Núcleo, ... ).

A seguir descrevem-se os suportes que o núcleo deverá oferecer para os propósitos do trabalho.

### VII.2.3.1 SUPORTE PARA DETECÇÃO E ELIMINAÇÃO DE MENSAGENS ÓRFÃS

Conforme descrito em VI.4.2.5. B), mensagens órfãs serão detectadas por meio do esquema de numeração seqüencial de mensagens e que tal suporte deverá ser oferecido pelo núcleo, em sua camada de "software" de portos. Para tanto, criou-se um processo que foi denominado de gerente de recepção de mensagens, o qual não deve ser interrompido, cuja primeira função, quando acordado pelo "software"

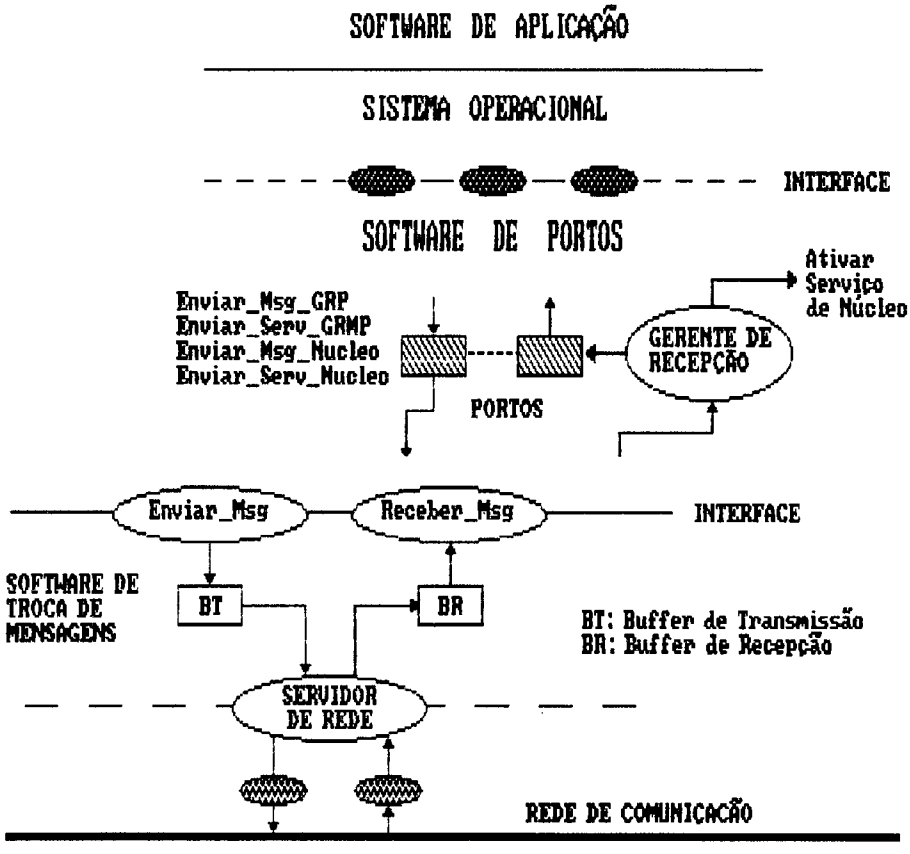



Figura VII.6: Esquema Ilustrativo do Núcleo de um nodo operador ou do nodo especial

de rede, é a de receber a mensagem remota e executar as seguintes tarefas:

- extrair o número de seqüência da mensagem e a identificação do destino;
- verificar se esse número é maior que o número da última mensagem recebida por aquele destino ( retida em uma tabela );
- se for, então, passar para a segunda etapa que é a de analisar o tipo da mensagem e colocar em execução a tarefa adequada, como será descrita no item seguinte a este;
- se não for, ignorar a mensagem ( descartar ).

A figura VII.6 ilustra a esquematização do software de portos onde residirão o gerente de recepção e as primitivas descritas no item anterior, como também, o software de troca de mensagens que juntos compõem algumas das tarefas do núcleo de um nodo operador, inclusive daquele nodo especial.

### VII.2.3.2 SUPORTE PARA DETECÇÃO E ELIMINAÇÃO DE COMPUTAÇÕES ÓRFÃS

No item VI.4.2.5. C), descreveu-se o suporte para detecção e eliminação de computações órfãos, para o mecanismo para comunicação e sincronização entre processos em questão. De acordo com as hipóteses lá estabelecidas, somente nas comunicações, segundo o esquema de "rendez-vous" estendido, podem surgir tais eventos indesejáveis. Ainda naquele item, descreveu-se as tarefas que as primitivas Receber_PEDRA e Responder_PEDRA, devem executar como o referido suporte. Nesse, aparece a necessidade de se enviar mensagens para o GRMP, para que este execute serviços como parte integrante desse suporte. Apenas para lembrar, os serviços são:

- marcar um ponto de recuperação e sinalizá-lo para evitar que ele marque pontos de recuperação dentro da extensão do "rendez-vous";
- solicitar o descarte da mensagem salva na cópia do processo, como também, para decrementar o número de mensagens já enviadas e recebidas da cópia do processo fonte;
- verificar se o ponto de recuperação foi devidamente estabelecido;
- solicitar a recuperação do processo para o ponto de recuperação pedido;

## VII.2.3.3 SUPORTE PARA OS MECANISMOS PARA RECUPERAÇÃO E MIGRAÇÃO DE PROCESSOS

### SUPRIMENTO DE INFORMAÇÕES PARA O GRMP:

Como visto no item VI.4.2.5.A), todas mensagens trocadas entre os processos do sistema, devem ser devidamente enviadas também para o GRMP, como também isso é tarefa de cada primitiva do "software" de portos de emissão de mensagens. Assim, as primitivas Enviar_PSDRA, Responder_PEDRA, Enviar_PSDCA, Enviar_PSS e Enviar_PD, devem executar as tarefas naquele item descrito, ou sejam:

(a) para as primitivas bloqueantes:

- quando a mensagem é para um processo local ( processos emissor e receptor residem em um mesmo nodo ):
  . enviar primeiro a mensagem para o GRMP e esperar a confirmação de recepção;
  . se confirmada, então, colocar a mensagem no porto de entrada destino ( porto de processo destino );
  . caso contrário, levantar tal exceção;
- quando a mensagem é para um processo remoto ( o processo destino é remoto ):
  . enviar primeiro a mensagem para o GRMP e esperar a confirmação da recepção;
  . se confirmada, então, enviar a mensagem para o porto de entrada destino ( porto de processo destino remoto ) e esperar confirmação;
  . caso confirmada, então, o processamento segue seu caminho normal;
  . caso não chegue a confirmação de recepção por parte do GRMP, deve ser levantada uma exceção e, conseqüentemente, não sendo enviada a mensagem para o porto de entrada destino.
  . caso não chegue a confirmação de recepção por parte do porto de entrada do processo destino, então, deve ser enviada uma mensagem para o GRMP descartar a mensagem salva para aquele processo cópia e terminar a operação levantando uma exceção.

(b) para as primitivas não bloqueantes:
- quando a mensagem é para um processo local ( processos emissor e receptor residem em um mesmo nodo ):
  . enviar primeiro a mensagem para o GRMP e esperar a confirmação de recepção;
  . se confirmada, então, colocar a mensagem no porto de entrada destino ( porto de processo destino );
  . caso contrário, não colocar a mensagem no porto de entrada destino ( porto de processo destino );
- quando a mensagem é para um processo remoto ( o processo destino é remoto ):
  . enviar primeiro a mensagem para o GRMP e esperar a confirmação da recepção;
  . se confirmada, então, enviar a mensagem para o porto de entrada destino ( porto de processo destino remoto ) e esperar confirmação;
  . caso confirmada, então, o processamento segue seu caminho normal;
  . caso não chegue a confirmação de recepção por parte do GRMP, então, não enviar a mensagem para o porto de entrada destino.

## SERVIÇOS DO NÚCLEO SOLICITADOS PELO GRMP

Descreve-se, agora, a parte referente aos serviços que o núcleo de um nodo operador deve oferecer para o mecanismo para recuperação e migração de processos: serviços solicitados pelo GRMP para recuperar ou migrar um processo naquele nodo. Os serviços são:

(a) se a mensagem for comum, então, acordar o processo que a está esperando. Lembre-se que toda mensagem deve carregar o identificador do porto e do processo destino;

(b) se a mensagem é um comando proveniente do GRMP, então:
  (1) se é para marcar um ponto de recuperação de um determinado processo:
    - se o referido processo está em execução, colocá-lo no estado de suspenso para marcação de ponto de verificação;
    - transmitir para o GRMP, todas as informações necessárias para tal;

(2) se a mensagem é para recuperar um processo ( migração - veja item IV.4 - seguido de informações de recuperação ):
- alocar um espaço de memória para colocar o código do processo a ser recuperado enviado por meio de mensagens, como também a pilha de trabalho e ajustar os endereçamentos de relocação;
- criar um descritor de processo e preenchê-lo com os dados contidos na mensagem enviada;
- criar os portos do processo a ser recuperado e atualizar os portos de entrada com as mensagens já recebidas, porém, ainda não consumidas;
- atualizar as tabelas de conexões dos portos;
- ajustar o contador de controle de mensagens já enviadas e consumidas. Essa informação é usada pelas primitivas de envio de mensagens dos portos de saída;
- enviar mensagem para o GRMP para confirmação do serviço executado;
- observação: na realidade, para cada parte do serviço executado, é enviado uma mensagem do tipo acima;
- se tudo correu bem, então, colocar o referido processo como pronto para execução;
- se ocorreu algum problema, então, abortar a tarefa e desfazer tudo que foi feito;

(3) se a mensagem é para eliminar processo:
- verificar se o referido processo está no estado de em migração;
- se não estiver, então, acusar a exceção para o GRMP;
- senão, eliminar todas as informações referentes a esse processo e solicitar ao gerente de memória a incorporação desse espaço como memória disponível;
- por fim, sinalizar o GRMP de que tudo correu bem;

(4) se é para atualizar conexões dos portos, então, atualizar as tabelas de conexões dos portos. Isso porque, uma vez feita a recuperação de um processo ( que por hipótese, é sempre precedida por uma migração de processo ), devem-se atualizar as tabelas de conexões de todos os núcleos dos nodos operadores.

Além desse suporte, o núcleo deverá oferecer outro para a configuração e reconfiguração dinâmica do sistema, que a seguir será descrito.

## VII.2.4 PRIMITIVAS PARA CONFIGURAÇÃO E RECONFIGURAÇÃO DINÂMICA DO SISTEMA

A configuração inicial do sistema de "software" é a fase da sua implantação, onde define-se a sua arquitetura em termos da distribuição de seus processos pelos nodos operadores do "hardware" do Sistema Distribuído e pelo estabelecimento da rede lógica de comunicação representada pelas conexões dos portos dos processos.

Percebe-se que o ideal seria que o ambiente para desenvolvimento de "software" oferecesse uma linguagem de alto nível para programação distribuída e respectiva linguagem para configuração. Algumas pesquisas objetivaram esse propósito como, por exemplo, o projeto CONIC, devida a KRAMER e MAGEE ( 1985 ) e KRAMER et alii ( 1983, 1984 ) ou a linguagem de programação Ester ( LOPES et alii, 1988 ); enfim, as encontradas nas referências citadas no capítulo IV. Notou-se, entretanto, que essas duas facilidades são bastantes complexas para serem desenvolvidas como ferramentas de uso geral. Assim e diante de não ser este o objetivo do trabalho e não se dispor de tais recursos, procura-se descrever um suporte para tal e as respectivas primitivas.

Assim sendo, levantam-se as funções referentes à configuração de um sistema, considerando a linguagem de programação utilizada; a Modula-2. A cada função necessária, descrevem-se as primitivas, se for o caso.

A linguagem Modula-2 oferece como conceito de processos a co-rotina. Um conceito de muito baixo nível para os propósitos do trabalho. Assim, deverão ser criados processos por meio das primitivas já definidas; a Criar_Processo.

Diante da utilização do conceito de portos locais, cada processo é responsável pela criação de seus portos. Conseqüentemente, por exemplo, um processo após criar um porto de saída não poderá imediatamente se utilizar dele para enviar mensagens, pois, primeiro é preciso conectá-lo a um porto de entrada compatível. Para isso, é preciso conhecer o porto de entrada desejado pertencente a um

processo destino das mensagens. Assim, os processos ficam impossibilitados de eles próprios estabelecerem as conexões, senão entram em conflito com o conceito de portos locais, como também, haveria a necessidade de especificar a ordem inicial de execução dos processos, para evitar que um processo solicite a conexão com um que ainda nem foi criado. Uma forma para resolver esse problema foi a de criar um par de primitivas, a saber:

- **Esperar_Sinal_Partida:** primitiva que suspende o processo que a executa, colocando-o no estado a espera de um sinal de partida e envia um sinal para o configurador. Tal primitiva deve ser usada por um processo após a criação de todos os seus portos e antes de iniciar o envio ou recepção de mensagens;
- **Disparar_Sistema:** primitiva que envia o sinal de partida para todos os processos suspensos à espera desse sinal. Essa primitiva deve ser usada pelo programa Configurador do Sistema.

Uma outra foram seria fazer com que as primitivas de envio ou de recepção de mensagens, executassem tarefas com esse objetivo, ou seja, um processo ao solicitar o envio de uma mensagem, a respectiva primitiva primeiro verificaria se o porto está conectado com algum outro. Se estiver, então, ela continua com suas tarefas seguintes. Caso contrário, ela suspende o processo à espera de um sinal que deverá ser enviado pela primitiva de conexão. À primeira vista, esse esquema se mostra interessante, observa-se, porém, que ele é restritivo, pelo menos quanto à necessidade da especificação da ordem inicial de execução, como já mencionado, para permitir as conexões uma para vários. É um caso a ser estudado.

Seguindo o esquema em descrição, o programa Configurador do Sistema, primeiro fica à espera de uma quantidade bem definida de sinais emitidos pelas primitivas Esperar_Sinal_Partida. A quantidade de sinais é bem definido porque ele deve saber a quantidade de processos componentes do sistema. Em seguida, ele pode executar sua tarefa de estabelecimento das conexões dos portos de acordo com a especificação da rede lógica de comunicação, ou seja, a da especificação do Sistema Operacional Distribuído. Por último, o Configurador deve executar a primitiva Disparar_Sistema, naturalmente, se tudo até aqui não surgiu nenhum problema. Nota-se, então, que para isso, o Configurador precisa de primitivas para:

- **Esperar_Sinais:** primitiva para suspender o processo até que chegue um determinado número de sinais especificados como parâmetro. Primitiva para ser usada pelo Configurador.
- **Disparar_Sistema:** já descrita;
- **Conectar_Porto:** já descrita ( item VII.2.2 );

Quanto à reconfiguração dinâmica do sistema, que pode se fazer necessário quando da recuperação de um processo ou da migração de processos por questões de decrescimento ou crescimento incremental do sistema, ou por questão de balanceamento dinâmico de cargas, esses mecanismos necessitarão, no caso já descritas, de primitivas para:

- **Desconectar_Porto;**
- **Conectar_Porto;** e
- **Eliminar_Processo;**

Dessa forma, o esquema de configurador do "software" do Sistema Distribuído é a seguinte:

(1) todos os processos devem ser programados de forma a criar primeiro todos os seus portos e em seguida solicitar o serviço da primitiva Esperar_Sinal_Partida;
(2) criar todos os processos do sistema - isto porque a linguagem Modula-2 oferece apenas o conceito de co-rotina;
(3) definir uma configuração mínima: especificação da quantidade mínima de nodos operadores, quais serão eles, suas características físicas e a distribuição dos processos nesses nodos;
(4) estabelecer todas as conexões dos portos e, com isso, completar a configuração mínima;
(5) definir uma estrutura de dados para manter em arquivo a configuração mínima, a que realmente está em operação e as que estiveram em operação, as quais devem ser eliminadas quando iniciado pela primeira vez o sistema. Tabelas a respeito de outras configurações mínimas possíveis ( combinações de distribuição de processos, mantendo-se o número mínimo de nodos operadores e suas características básicas );
(6) executar a primitiva Esperar_Sinais;
(7) executar Disparar_Sistema;

Na realidade, para o desenvolvimento de um Configurador de Sistema e com as restrições existentes, é interessante que sejam criados ferramentas que auxiliem a execução dessa tarefa, por exemplo:

- procedimento para visualizar graficamente a rede lógica de comunicação;

- procedimento para verificar se algum porto continua desconectado;
- procedimento para verificar as características de um porto;
- procedimento para verificar as características de um nodo operador;
- etc.;
- tudo isso controlado por uma interface gráfica;

A seguir descrevem-se sugestões a respeito da iniciação de Sistemas Operacionais Distribuído.

## VII.3 UM SUPORTE PARA INICIAÇÃO DE SISTEMA OPERACIONAL DISTRIBUÍDO COM RECONFIGURAÇÃO DINÂMICA DE PROCESSOS

A iniciação de um Sistema Operacional é a sua colocação em execução. De um modo geral, um computador para aplicações genéricas, é construído para facilitar a implantação de sistema de "software" que especifica os seus propósitos de utilizações. Para isso, é comum, que uma pequena parte de sua memória principal seja composta de elementos não voláteis e contra escrita, quando em operação normal e, já possua o mínimo de recursos de "software" necessários para poder controlá-lo por meio de ações externas. Geralmente, essas ações se resumem na simples colocação de um disco flexível em um determinado acionador de discos que contenha o Sistema Operacional completo, ou apenas uma primeira parte, que por sua vez, solicita a colocação de outros e/ou o acionamento de outros dispositivos periféricos, como unidades de fita magnética, por exemplo. Resumindo, o computador, uma vez ligado, deve acionar automaticamente sua primeira tarefa que coloca em execução outras, até o Sistema Operacional estar completamente definido, de acordo com a configuração desejada ou possível. Alguns microcomputadores pessoais, colocam como sua primeira tarefa um monitor residente que cria um pequeno ambiente para programação em "assembly" ( linguagem montadora ). De qualquer forma, sempre deverá existir, como parte integrante de um computador, um elemento que dê o primeiro passo na direção da colocação do "software" desejado pelos seus usuários. Esse passo é comumente denominado de "bootstrap loader" primário ( carregador primário ) e os recursos básicos necessários como suporte para ele são denominados de BIOS ( Basic Input Output System ), mencionados no início como mínimo de recursos de "software".

Assim considerando, para o Sistema Distribuído em questão, admitir-se-á que cada nodo operador possua uma memória não volátil ( memória do tipo ROM - Read Only Memory ), que conterá além do programa verificador de suas condições e capacidades físicas, como tamanho de memória volátil ( tipo RAM - Random Acess Memory ) disponível para programas, se possui memória secundária e interfaces para outros dispositivos periféricos, como impressora, saída serial, etc., dispõe-se de um programa carregador primário com as seguintes funções:

- verificar se o nodo operador está conectado ou não à rede de comunicação;
- se estiver, então, carregar o núcleo multitarefas completo ( com as informações sobre a conexão dos portos e etc. ), através do Configurar do Sistema;
- carregar os processos que lhe cabe do Sistema Operacional Distribuído através do Configurador do Sistema;
- feito isso, esperar o sinal de partida;
- se o nodo não estiver conectado à rede de comunicação, então, carregar o Sistema Operacional de uma de suas entradas principais que pode serem: acionador A de disco ou de sua memória secundária. Isso pensando que os nodos são microcomputadores do tipo IBM-PC XT ou AT ou 386.

O carregador primário do nodo operador especial terá diferentes atribuições que aquele, elas são:

- carregar de sua memória secundária o núcleo multitarefas, o processo GRMP e todos os outros componentes do sistema, como o sistema de arquivo, gerente de memória, etc.;
- verificar se existe pelo menos uma configuração definida: o Sistema Operacional Distribuído;
- se existir, então, verificar qual delas é possível, no momento, ser colocado em operação;
- se não existir nenhuma configuração diretamente possível, então, gerar uma a partir da configuração mínima;
- feito isso, eliminar as demais, salvo a mínima;
- ativar um "software" que terá a função de distribuir o núcleo e os processos para os nodos operador do Sistema Distribuído, de acordo com a configuração determinada;
- por fim, disparar o sistema, emitindo para todos os nodos operadores ativos, um sinal de partida;

- caso não exista nenhuma configuração ( nem a mínima ), então, avisar o operador desse fato e esperar ações externas.

Observa-se que essa é uma esquematização de iniciação de Sistemas Operacionais Distribuídos, não se teve a pretensão de definir a única possível, como a ideal.

# CAPÍTULO VIII

## CONCLUSÕES

### VIII.1 ASPECTOS GERAIS DO TRABALHO

Este trabalho abordou um dos sérios problemas enfrentados na construção de Sistemas Operacionais Distribuídos: a confiabilidade - no sentido de serem tolerantes a falhas, visando aumentar a taxa de disponibilidade, uma das grandes metas, ou justificativas, na concepção de Sistemas Operacionais Distribuídos.

Nesse contexto, diante de sua grande extensão, abrangendo desde a concepção do "hardware" até o "software" que faz interface com os usuários, restringiu-se a abordagem na análise e no projeto de suporte para Sistemas Operacionais Distribuídos, consistindo de um mecanismo para comunicação e sincronização entre processos baseado em portos, com suporte para detecção e eliminação de órfãos e de mecanismos para recuperação e migração de processos. Objetivou-se, com isso, contribuir com a pesquisa na construção de Sistemas Operacionais Distribuídos, apresentando soluções alternativas para os problemas relacionados com a confiabilidade de mecanismos para comunicação e sincronização entre

processos e com a reconfiguração dinâmica de sistema. Tais mecanismos são importantes para evitar que a presença de defeitos em componentes processadores do Sistema Distribuído provoque um colapso com perda de grande parte de trabalho útil feito até então.

Para o desenvolvimento do projeto, tomou-se como base a existência de uma infra-estrutura, consistindo, basicamente, de uma rede de comunicação e de um protocolo de comunicação fim a fim entre os núcleos dos nodos operadores, que apenas detectasse e ignorasse mensagens corrompidas. Dessa maneira, o suporte poderia servir de bancada de trabalho para o desenvolvimento de Sistemas Operacionais Distribuídos com a característica de serem reconfiguráveis dinâmicamente, de forma relativamente independente da infra-estrutura a nível de subsistema de comunicação.

Foi admitida a existência de uma camada de "software", no núcleo de cada nodo operador componente do Sistema Distribuído, que oferecesse as primitivas básicas para poder estabelecer as comunicações entre os núcleos dos nodos operadores. Isso permite que os mecanismos de suporte em questão sejam portáteis ou reutilizáveis, ou ainda adaptados sem maiores complicações. Isso facilita uma análise experimental de desempenho desses mecanismos, de forma comparativa, levando em conta várias topologias de rede de comunicação.

Para a escolha dos mecanismos mais apropriados, foi feita uma vasta revisão bibliográfica relacionadas com o assunto de tolerância a falhas para dar suporte à construção de Sistemas Operacionais Distribuídos com Reconfiguração Dinâmica de Processos. Nessa revisão incluíram-se muitas técnicas para tratamento de falhas que, apesar de não terem sidos diretamente usadas na proposta da tese, contribuíram para se formar uma boa base a respeito de mecanismos para tolerância a falhas de "software" e de "hardware". Por exemplo, pode-se citar o bloco de recuperação, a programação N-versões, etc.

A opção pelo conceito de portos locais, como mecanismo para comunicação e sincronização entre processos, deveu-se aos seguintes fatores:

- facilidade para a programação modular, devido a não necessidade por parte de um processo de conhecer os identificadores daqueles com quem ele quer se comunicar;
- facilidade para contruir de forma dinâmica a rede lógica de comunicação entre os processos do sistema, facilitando a tarefa de reconfigurar dinâmicamente o sistema; e

- facilidade para a implementação de várias formas de comunicação e, com isso, abrangendo uma gama razoavelmente grande de aplicações;

Procurou-se tornar textualmente claras as intenções de uma comunicação, visando facilitar as tarefas de verificação, correção e manutenção do sistema. Para isso, as primitivas foram nomeadas de forma a explicitarem as suas funções básicas, como por exemplo, se a comunicação é assíncrona e um para um, ou assíncrona e um para vários, etc. Além disso, por questões de robustez contra erros de utilização no estabelecimento de uma comunicação por parte dos programadores, os portos são criados com a associação do tipo de mensagem que eles podem reter para emissão ou recepção. Assim, em tempo de execução, tanto as primitivas de conexão como as de comunicação, têm todas as condições de verificarem a compatibilidade da conexão entre portos solicitada, como também entre o tipo da mensagem a ser transmitida ou recebida e o tipo de mensagem suportada pelo porto utilizado.

Também, as primitivas, apesar de não terem sido implementadas de forma tipificadas, poderiam assim serem construídas, permitindo que muitos erros de programação fossem detectados em tempo de compilação. Apesar disso parecer bastante interessante, à primeira vista, pode ser que essa especialização de primitivas acabe complicando a sua utilização pelos processos do sistema, na medida em que passa a ser necessário um conjunto específico de primitivas para criar porto e para operá-lo para cada tipo de mensagem. Isso deve ser melhor analisado, tanto no aspecto de tipificação, como na quantidade de memória necessária para reter os códigos dessas primitivas, pois, quanto maior o núcleo, menor fica o espaço de memória disponível para os programas dos usuários. Porisso, considerou-se mais adequado, no momento, implementar esse mecanismo de forma a deixá-lo em um nível um pouco mais baixo e, portanto, mais flexível. Procurou-se assim, manter o mínimo de primitivas necessárias e suficientes para o estabelecimento das formas básicas de comunicação: um para um; um para vários; e vários para um. Assim, ficaram para os usuários os cuidados a serem tomados quanto à criação dos portos com os tipos de mensagens que eles desejam associar e à formatação e a interpretação das mensagens.

Ainda em termos de robustez, porém, agora, no sentido de confiabilidade funcional do mecanismo, foi permitido na especificação de cada primitiva para comunicação síncrona, o número máximo de tentativas para o término de sua operação, como também, o tempo limite

para cada uma delas. Isso foi feito para que esse tipo de primitiva possa competir com falhas de comunicação, como aquelas restritas às provenientes de defeitos de nodos operadores e de comunicação e perdas de mensagens por corrupção. Também foi criado um suporte para a detecção e eliminação de órfãos, muito importante para as comunicações na forma de "rendez-vous" estendido.

Para a eliminação de computações órfãs, foi utilizado o mecanismo para recuperação de processos, devidamente estruturado para incorporar essa tarefa, também foi feita a migração pura de processos, além de suas atribuições normais de recuperação de processos, quando da detecção de colapso de nodos operadores. Para isso, acabou-se impondo a restrição de que todos os processos devem ser determinísticos, ou seja, na recuperação de um processo não será desfeita nenhuma operação fora de sua memória principal já realizada; ele será simplesmente colocado em um estado anterior à detecção da ocorrência de uma falha, e todas as comunicações feitas deste ponto, até o ponto em que estava, serão devidamente simuladas. Dessa forma, a eliminação de uma computação órfã, ficou restrita às mesmas condições.

Portanto, caso os efeitos de uma computação órfã devam ser desfeitos, por suas operações não serem idempotentes, haverá a necessidade da utilização de um mecanismo para isso, como por exemplo, o "cache" de recuperação do esquema de bloco de recuperação. A incorporação de tal tipo de mecanismo, no suporte para detecção e eliminação de computações órfãs, não é interessante pelo seguinte motivo: sobrecarregaria demais o sistema com computações não úteis, pois muitas operações idempotentes seriam desfeitas, quando da restauração do estado do processo. Dessa forma, é preferível deixar essa tarefa no local mais apropriado, ou seja, nos processos, que para isso poderiam usar o bloco de recuperação, ou uma outra estrutura de programação que possua essa característica de desfazer operações.

Quanto à parte prática do trabalho, toda ela está sendo implementada na linguagem Modula-2 em um único microcomputador do tipo IBM PC XT/AT/386. Desse modo, o ambiente de rede está sendo simulado da seguinte forma:

- o microcomputador, onde tudo é executado, é considerado como um nodo do Sistema Distribuído;
- os processos contidos em nodos remotos estão sendo simulados através da tabela dos processos distribuídos ( figura VII.1 );

- o tempo para a comunicação entre dois processos, não residentes em um mesmo nodo, está sendo simulado pela colocação de um atraso no processo receptor; e
- o surgimento de falhas do subsistema de comunicação está sendo simulado, por enquanto, de modo bem determinado pelos próprios processos. Após uma boa bateria de testes, se está pensando em simulá-lo por meio de um gerador de aleatório que deverá ser consultado pelo mecanismo de comunicação entre nodos operadores, para verificar se ele deverá falhar ou não;

O ambiente suporte, constituído do núcleo multitarefas para os teste dos mecanismos em questão, está todo implementado, faltando inserir o gerador de aleatórios para a simulação da ocorrência de falhas. Quanto aos mecanismos propriamente ditos, encontra implementado, e em fase de testes finais, o mecanismo para comunicação e sincronização entre processos baseado em portos locais, incluindo a parte de detecção e eliminação de mensagens órfãs e detecção de computações órfãs. Encontram-se esquematizados o suporte e os mecanismos para recuperação e migração de processos e, consequentemente, a eliminação de computações órfãs.

Apenas para dar uma idéia da dimensão do núcleo em questão, listam-se a seguir, os dados referentes a ele, com e sem a imposição de características de tolerância a falhas:

1) quanto a parte implementada:
   a) dimensões dos módulos fontes:
      - sem as características de tolerância a falhas:
        *148.474 bytes, correspondentes a 4.388 linhas*
      - com as características de tolerância a falhas:
        *193.176, bytes correspondentes a 5.740 linhas*
   b) dimensões dos módulos testes objetos:
      - sem as características de tolerância a falhas:
        *66.683 bytes*
      - com as características de tolerância a falhas:
        *86.457 bytes*
2) quanto ao acréscimo devido ao suporte para recuperação e migração de processos:
   - essa parte encontra-se esquematizada, da qual projeta-se que o acréscimo de "software" ficará em torno de 15%.

É perceptível que o núcleo com o suporte para recuperação e migração de processos, da forma proposta, incluindo-se a arquitetura do subsistema de comunicação adotada, impõe um custo natural ao seu desempenho. Isso se deve ao fato de que, toda mensagem transmitida de um processo a um outro, deverá também ser enviada ao GRMP, o que mostra a aderência desse suporte, ao tipo de arquitetura do meio de comunicação subjacente. Assim sendo, tal custo não deverá se constituir em um problema que inviabilize tal suporte de recuperação e migração de processos, pois ele poderá ser minimizado, senão até eliminado, ao se adotar um subsistema de comunicação mais adequado. Por exemplo, um composto de linha de comunicação para a transmissão normal de mensagens e uma outra para fins específicos para o referido mecanismo. Dessa forma, a dupla transmissão poderá ser feita em paralelo. Esse é um caso a ser pesquisado sobre o projeto CENTAURO, listado como pesquisa futura a curto e médio prazo, logo adiante.

Finalizando, acredita-se que a parte prática desse projeto de núcleo, deverá servir como uma boa bancada de trabalho para o desenvolvimento de Sistemas Operacionais Distribuídos reconfiguráveis dinamicamente.

## VIII.2 CONTRIBUÍÇÕES DO TRABALHO

A contribuição principal refere-se ao estudo e implementação de um mecanismo de comunicação e sincronização entre processos baseado no conceito de portos locais e com suporte para detecção e eliminação de órfãos. Inclui-se nessa contribuição, a integração de um mecanismo para recuperação de processos, necessário para a eliminação de computações órfãs.

Como uma consequência, praticamente natural, a integração de um mecanismo para migração de processos ao mecanismo para recuperação de processos, tornou-se uma outra contribuição no sentido de que eles pudessem, também, serem utilizados como suporte para a recuperação de processos na ocorrência de falhas de seus nodos operadores. Isso, independentemente dos processos estarem ou não em comunicação.

Complementando a contribuição, descreveu-se no capítulo VII a implementação desses mecanismos para constituir-se em uma bancada de trabalho para o desenvolvimento de Sistemas Operacionais Distribuídos Tolerantes a Falhas de comunicação e de colapso de nodos operadores.

## VIII.3 PESQUISAS FUTURAS

A seguir são listadas as pesquisas futuras para dar continuidade a este trabalho, como também aquelas dentro do contexto de tolerância a falhas em Sistemas Distribuídos.

### A) PESQUISAS A CURTO E MÉDIO PRAZO

- desenvolver os mecanismos para recuperação e migração de processos nesse mesmo esquema, porém, considerando a rede de comunicação com conexão ponto a ponto formando um laço. Posteriormente procurar-se-á descentralizar o GRMP.
- complementação do mecanismo para recuperação e migração de processos para incorporar um mecanismo de "cache" de recuperação, visando posterior aplicação do conceito de ações atômicas para a estruturação de Sistemas Operacionais Distribuídos.
- impor aspectos de tolerância a falhas no Projeto CENTAURO de Computação Paralela do Departamento de Computação da UFSCar, como uma aplicação dessas técnicas. No projeto CENTAURO está sendo desenvolvido um sistema paralelo composto de um conjunto de processadores fracamente conectados, através de barramentos paralelos, organizados hierárquicamente, formando vários "clusters". Além da importância visível da necessidade de construção de "software" tolerante a falhas nesse sistema, sua arquitetura que contém redundância do meio de comunicação mostra-se, à primeira vista, bastante aderente aos esquemas dos mecanismos para recuperação e migração de processos.
- um ponto a ser analisado futuramente é quanto à frequência ideal de estabelecimento automático de pontos de recuperação em cada processo do sistema. Para fins de primeira implementação foi definido como a cada dez ( 10 ) transmissões e/ou recepções de mensagens por um processo. À primeira vista, nota-se que a frequência não deve ser alta, pois, caso a probabilidade do surgimento de falhas seja alta, de duas uma, ou o custo em termos de processamento é plenamente justificável perante o custo do "hardware", ou é preferível melhorar a confiabilidade do "hardware".

## B) PESQUISAS FUTURAS DENTRO DO CONTEXTO DE TOLERÂNCIA A FALHAS

- estruturação de Sistema Operacional Distribuído em grupos de processos, como está sendo implementado o GRMP, visando minimizar tanto o custo de salvamento de pontos de recuperação, como também, o custo de comunicação. O artigo de LIANG et alii ( 1990 ) é uma boa referência nessa direção.
- estudos sobre Sistemas Distribuídos construídos com replicação de componentes ( que de um modo geral, não estão livres de algum defeito ) para serem utilizados no esquema de NMR ( N-Modular Redundancy ) para impor confiabilidade. Referências tais como: SHRIVASTAVA ( 1986 ), MANCINI e SHRIVASTAVA ( 1986, 1987 ), MANCINI e PAPPALARDO ( 1987, 1988a, 1988b ), MANCINI e KOUTNY ( 1986 ), EZHILCHELVAN ( 1987 ), KOUTNY e MANCINI ( 1987 ), EZHILCHELVAN et alii ( 1988, 1989 ), constituem-se em um bom começo, pois variam desde as primeiras idéias a respeito, passando pelas preocupações com a análise e desempenho e chegando aos problemas dos mecanismos para comunicação. Nesse contexto, encaixa-se o projeto CENTAURO, no qual pode-se pensar em tornar cada "cluster" altamente confiável, através desse esquema NMR ou de programação N-versões e com a aplicação do conceito de grupo de processos, fazendo com que cada um possua o seu GRMP; isso à primeira vista parece que deverá diminuir bastante o tráfego de comunicações entre "clusters", o que deverá melhorar o desempenho do sistema, como também, o esforço de implementação.

# REFERÊNCIAS BIBLIOGRÁFICAS

ALFORD, M.W.; ANSART, J.P.; HOMMEL, G.; LAMPORT, L.; LISKOV, B.; MULLERY, G.P. E SCHNEIDER, F.B. ( 1985 ), "Distributed Systems - Methods and Tools for Specification ( An Advanced Course )", Lectures Notes in Computer Science 190, Ed. by M.Paul and H.J. Siegert, Springer Verlag, Berlin 1985.

ANDERSON, T. e Lee, P.A. ( 1979 ), "The Provision of Recoverable Interfaces", IEEE Digest Papers, FTCS-9, Madison, p.87-94, june 1979. ( reimpresso em SHRIVASTAVA, 1985 ).

ANDERSON, T. e KERR, R. ( 1976 ), "Recovery Blocks in Action: A System Supporting High Reliability", Proceedings of the 2nd Intl. Conf. on Software Engineering, october 1976, San Francisco, p. 447-457. ( reimpresso em SHRIVASTAVA, 1985 ).

ANDERSON, T. e Lee, P.A. ( 1981 ), "Fault Tolerance: Principles and Practice", Prentice-Hall International, Inc., London, 1981.

ANDERSON, T. e Lee, P.A. ( 1982 ), "Fault Tolerance Terminology Proposals", IEEE Digest of Papers, FTSC-12, Santa Monica, p.29-33, june 1982. ( reimpresso em SHRIVASTAVA, 1985 ).

ANDERSON, T. e Randell, B. ( 1979 ), "Computing Systems Reliability ( Collected Papers )", Hamilton Printing Company, Rensselaer, New York, 1979.

ANDERSON, T.; LEE, P.A. e SHRIVASTAVA, S.K. ( 1978 ), "A Model of Recoverability in Multilevel Systems", IEEE Transactions on Software Engineering, vol. SE-4, n.6, p.486-494, november 1978. ( reimpresso em SHRIVASTAVA, 1985 ).

ANDREWS, G.R. ( 1981 ), "Synchronizing Resource", ACM Transactions on Programming Languages and Systems, vol 3 (3), october 1981.

ANDREWS, G.R. ( 1982 ), "A Distributed Programming Language SR - Mechanisms, Design and Implementation", Software-Practice and Experience, vol 12, 719-753, 1982

ANDREWS, G.R. e SCHNEIDER, F.B. ( 1983 ), "Concepts and Notations for Concurrent Programming", Computing Surveys, vol 15(1), march 1983.

BACON, J. ( 1981 ), "An Aproach to Distributed Software Systems", Operating Systems Review, vol 15(4), october 1981.

BALZER, R.M. ( 1971 ), "PORTS - A Method for Dynamic Interprogram Communication and Job Control", Spring Joint Computer Conference, 1971.

BARNES, J.G.P. ( 1980 ), "An Overview od Ada", Software-Practice and Experience, vol 10, 851-887, 1980.

BIRREL, A.D. ( 1985 ), "Secure Communication Using Remote Procedure Calls", ACM Transactions on Computer Systems, vol 3 (1), 1-14, february 1985.

BORG, A.; BAUMBACH, J. e GLAZER, S. ( 1983 ), "A Message System Supporting Fault Tolerance", Proceedings of 9th Symposium of Operating Systems Principles, p.90-99, october 1983.

BOS, J.V.D. ( 1980 ), "Comments on Ada Process Communication", ACM - SIGPLAN Notices, vol 15(6), june 1980.

BROWNBRIDGE, D.R.; MARSHALL, L.F. e RANDELL, B. ( 1982 ), "The Newcastle Connection or UNIXes of the World United", Software-Practice and Experience, vol 12, p.1147-1162, 1982. ( reimpresso em SHRIVASTAVA, 1985 ).

CAMPBELL, R.H. e KOLSTAD, R. B. ( 1980 ), "An Overview of Path Pascal's Design", ACM - SIGPLAN Notices, vol 15 (9), november 1980.

CHERRY, G.W. ( 1984 ), "Parallel Programming in ANSI Standard Ada", Reston Publishing Company, Inc., A Prentice-Hall Company, Reston, Virginia, 1984.

COOK, R.P. ( 1980 ), "*MOD - A Language for Distributed Programming", IEEE Transactions on Software Engineering, vol SE-6, N.6, november 1980.

CORSINI, P.; FROSINI, G. e LOPRIORE, L. ( 1984 ), "Distributing and Revoking Acess Authorizations on Abstract Objects: A Capability Approach", Software-Practice and Experience, vol 14 (10), 931-943, october 1984.

CRISTIAN, F. ( 1982 ), "Exception Handling and Software Fault Tolerance", IEEE Transactions on Computers, C-31, n.6, p.531-540, june 1982. ( reimpresso em SHRIVASTAVA, 1985 ).

CRISTIAN, F. ( 1990 ), "Understanding Fault-Tolerant Distributed Systems", IBM Thomas J. Watson Research Center, Yorktown Heights, New york 10598, june 1990.

DEITEL, H.M. ( 1984 ), "An Introduction to Operating Systems", Addison-Wesley Publishing Company, Inc., 1984

DENNING, P. ( 1976 ), "Fault-Tolerant Operating Systems", ACM Computing Surveys, vol 8 (4), december 1976.

DeREMER, F. e LEVY, P. ( 1979 ), "Summary of the Characteristics of Several Modern Programming Languages", ACM - SIGPLAN Notices, vol 14 (4), 28-46, 1979.

DIJKSTRA, E.W. ( 1968 ), "The Structure of the "THE" - Multiprogramming System", Communications of the ACM, vol. 11, n.5, may 1968.

EZHILCHELVAN, P.D. ( 1987 ), "Early Stopping Algoritms for Distributed Agreement under Fail-Stop, Omission, and Timing Fault Types", Technical Report Series, Computing Laboratory, University of Newcastle upon Tyne, n.235, may 1987.

EZHILCHELVAN, P.D.; MITRANI, I. e SHRIVASTAVA, S.K. ( 1989 ), "An Empirical Study of the Performance of Distributed Replicated System", Technical Report Series, Computing Laboratory, University of Newcastle upon Tyne, n.278, may 1989.

EZHILCHELVAN, P.D.; SHRIVASTAVA, S.K. e TULLY, A. ( 1988 ), "Construction Replicated Systems Using Processors with Point to Point Communications Links", Technical Report Series, Computing Laboratory, University of Newcastle upon Tyne, n.274, december 1988.

FELDMAN, J.A. ( 1979 ), "High Level Programming for Distributed Computing", Communications of the ACM, vol 22(6), june 1979.

FILMAN, R.T E. e FRIEDMAN, D.P. ( 1984 ), "Coordinated Computing - Tools and Techniques for Distributed Software", McGraw-Hill Book Company, Computer Science Series, 1984.

FORTIER, P.J. ( 1986 ), "Design of Distributed Operating Systems", Intertext Publications, Inc., McGraw-Hill, Inc., New York, N.Y., 1986.

GEHANI, N.H. e CARGILL, T.A. ( 1984 ), "Concurrent Programming in Ada Language: The Polling Bias", Software-Practice and Experience, vol 14(5),413-427, may 1984.

GENTLEMAN, W.M. ( 1981 ), "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", Software-Practice and Experience, vol 11, 435-466, 1981.

GHEZZI, C. e JAZAYERI, M. ( 1982 ), "Programming Language Concepts", John Wiley e Sons, Inc., N.Y. 1982

HANSEN, P.B. ( 1977 ), "The Architecture of Concurrent Programs", Prentice-Hall, Inc., Englewood Cliffs, N.J. 1977.

HANSEN, P.B. ( 1978 ),"Distributed Processes: A concurrent Programming Concept", Communications of the ACM, vol 21, n.11, november 1978.

HANSEN, P.B. ( 1982 ), "Programming a Personal Computer", Prentice-Hall, Inc., Englewood Cliffs, N.J., 1982.

HERLIHY, M.P. e McKENDRY, M.S. ( 1989 ) ( 1989 ), "Timestamp-Based Orphan Elimination", IEEE Transactions on Software Engineering, vol. 15, n.7, july 1989.

HOARE, C.A.R. ( 1974 ), "Monitors: An Operating System Structuring Concept", Communication of the ACM, vol 17, n.10, october 1974.

HOARE, C.A.R. ( 1978 ), "Communicating Sequential Processes", Communications of the ACM, vol 21, n.8, august 1978.

HOLT, R.C. ( 1983 ), "Concurrent Euclid, The Unix System, and Tunis", Addison-Wesley Publishing Company, Inc., 1983.

HOOPE, J. ( 1980 ), "A Simple Nucleous Written in Modula-2: A case Study", Software-Practice and Experience, vol 10, 697-706, 1980.

HORNING, J.J; LAUER, H.C.; MELLIAR-SMITH, P.M. e RANDELL, B. ( 1974 ), "A Program Structure for Error Detection and Recovery", Lectures Notes in Computer Science, vol. 16, p. 177-193, Springer-Verlag, 1974. ( reimpresso em SHRIVASTAVA, 1985 ).

JEGADO, M. ( 1983 ), "Recoverability Aspects of a Distributed File System", Software-Practice and Experience, John Wiley & Sons, Ltd., vol. 13, p.33-44, 1983. ( reimpresso em SHRIVASTAVA, 1985 ).

KIRNER, C. ( 1986 ), "Desenvolvimento de Suporte Básico para Sistemas Operacionais Distribuídos", Tese D.Sc., COPPE/UFRJ, Engenharia de Sistemas e Computação, 1986.

KIRNER, C. e MENDES, S.B.T ( 1988 ), "Sistemas Operacionais Distribuídos: Aspectos Gerais e Análise de sua Estrutura", Editora Campus, Rio de Janeiro, 1988.

KOHLER, W.H. ( 1981 ), "A Survey of Techniques for Synchronizations and Recovery in Descentralized Computer Systems ", Computing Surveys, vol 13 (2), june 1981.

KOUTNY, M. e MANCINI, L. ( 1987 ), "Synchronizing Events in Replicated Systems", Technical Report Series, Computing Laboratory, University of Newcastle upon Tyne, n.237, june 1987.

KRAMER, J. e MAGEE, J. ( 1985 ), "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, vol SE-11, n.4, april 1985.

KRAMER, J.; MAGEE, J.; SLOMAN, M.; TURDLE, K. e DULAY, N. ( 1984 ), "The CONIC Programming Language, version 2.4", Research Report, Department of Computing, Imperial College, 180 QeensGate, London SW7 2BZ.

KRAMER, J; MAGEE, J; SLOMAN, M. e LISTER, A. ( 1983 ), "CONIC: An Integrated Approach to Distributed Computer Control Systems", IEE Proceedings, vol 130, Pt. E, n.1, januay 1983.

LAMPSON, B.W. e REDELL, D.D. ( 1980 ), "Experience with Processes and Monitors in Mesa", Communications of the ACM, vol 23, n.2, february 1980.

LEBLANC, T. Jr; GERBER, R.H. e COOK, R.P. ( 1984 ), "The StarMod Distributed Programming Kernel", Software-Practice and Experience, december 1984.

LEE, P.A. ( 1978 ), "A Reconsideration of the Recovery Block Scheme", The Computer Journal, vol 21, N.4, p. 306-310, november 1978. ( reimpresso em SHIVASTAVA, 1985 ).

LEE, P.A.; GHNANI, N. e HERON, K. ( 1980 ), "A Recovery Cache for PDP-11", IEEE Transactions on Computers, C-29, n.6, p.546-549, june 1980. ( reimpresso em SHRIVASTAVA, 1985 ).

LIANG, L.; CHANSON, S.T. e NEUFELD, G.W. ( 1990 ), "Process Groups and Group Communications: Classifications and Requirements", IEEE Computer, february 1990.

LIN, K. e GANNON, J.D. ( 1985 ), "Atomic Remote Procedure Call", IEEE Transactions on Software Engineering, vol. SE-11, n.19, october 1985.

LINDEN, T.A. ( 1976 ), "Operating System Structures to Support Security and Reliable Software", ACM Computing Surveys, vol 8 (4), december 1976.

LISTER, A.M. ( 1983 ), "Fundamentals of Operating Systems", The Macmillan Press Ltd, 1983 ( second edition - first edition 1975 ).

LOGITECH, Inc. ( 1985 ), "Modula-2/86 : User's Manual", Logitech, Inc., 805 Veterans Blvd., Redwood City, CA 94063, USA 1985.

LOMET, D.B. ( 1977 ), "Process Structuring, Synchronization and Recovery Using Atomic Actions", Proceedings of the ACM Conference on Language Design for Reliable Software, vol 12 (3), march 1977. ( reimpresso em SHRIVASTAVA, 1985 ).

LOPES, A.B.; KAMADA, A.; GUIMARÃES, E.G. e COELLO, J.M.A. ( 1988 ), "Ster - Um Ambiente para Desenvolvimento de Software Tempo Real", DTIA-001/88, Instituto de Automação, Depart. de Eng. Integrada, Divisão de Software Básico em Tempo Real, 1988.

LORIN, H. e DEITEL, H.M. ( 1981 ), "Operating Systems", Addison-Wesley Publishing Company, Inc. Philippines 1981.

MAGUIRE, G.Q. e SMITH, J.M. ( 1988 ), "Process Migration: Effects on Scientific Computation", ACM - SIGPLAN Notices, vol.23(3), march 1988.

MANCINI, L. e KOUTNY, M. ( 1986 ), "Formal Specification of N-modular Redundancy", Technical Report Series, n.213, Computing Laboratory. University of Newcastle upon Tyne, may 1986.

MANCINI, L.V. e PAPPALARDO, G. ( 1987 ), "Resolving Nondeterminism in Distributed Redundant System", Technical Report Series, n.233, Computing Laboratory. University of Newcastle upon Tyne, February 1987.

MANCINI, L.V. e PAPPALARDO, G. ( 1988a ), "The Verification of Distributed Redundant Systems", Technical Report Series, n.253, Computing Laboratory. University of Newcastle upon Tyne, april 1988.

MANCINI, L.V. e PAPPALARDO, G. ( 1988b ), "Towards a Theory of Replicated Processing", Technical Report Series, n.258, Computing Laboratory. University of Newcastle upon Tyne, april 1988.

MANCINI, L.V. e SHRIVASTA, S.K. ( 1986 ), "Exception Handling in Replicated Systems with Voting", Technical Report Series, n.217, Computing Laboratory. University of Newcastle upon Tyne, may 1986.

MANCINI, L.V. e SHRIVASTA, S.K. ( 1987 ), "Failure Detection in Replicated Systems", Technical Report Series, n.238, Computing Laboratory. University of Newcastle upon Tyne, june 1987.

MANCINI, L.V. e SHRIVASTA, S.K. ( 1988 ), "Fault-Tolerant Reference Counting for Garbage Colletion in Distributed Systems", Technical Report Series, n.260, Computing Laboratory. University of Newcastle upon Tyne, june 1988.

MANNING, E.; LIVERSEY, N.J. e TOKUDA, H. ( 1980 ), "Interprocess Communication in Distributed Systems: One View", Proceedings IFIP, 1980.

MELLIAR-SMITH, P.M. e RANDELL, B. ( 1977 ), "Software Reliability: The Role of Programmed Exception Handling", Proceedings of the ACM Conference on Language Design for Reliable Software, vol 12 (3), march 1977. ( reimpresso em SHRIVASTAVA, 1985 ).

MERLIN, P.M. e RANDELL, B. ( 1978 ), "State Restoration in Distributed Systems", IEEE Digest of Papers, FTCS-8, Toulouse, p.129-134, june 1978. ( reimpresso em SHRIVASTAVA, 1985 ).

MITCHELL, J.G.; MAYBURY, W. e SWEET, R. ( 1979 ), "Mesa Language Manual", Xerox, Palo Alto Research Center, Systems Development Department, version 5.0 - april 1979.

NELSON, B.J. ( 1981 ), "Remote Procedure Call", Ph.D. Thesis, Departament of Computer Science, Carnegie-Mellow University, may 1981.

NELSON, V. ( 1990 ), "Fault-Tolerant Computing: Fundamental Concepts", IEEE Computer Society, July 1990.

OLIVEIRA, W. ( 1991 ), "Núcleo de um Sistema Operacional para uma Máquina Paralela", Tese de Mestrado, Departamento de Computação, UFSCar, abril 1991.

PAKER, Y. e VERJUS, J.-P. ( 1983 ), "Distributed Computing Systems - Synchronization, Control and Communication", Academic Press Inc. Ltda, London, 1983.

PANZIERI, F. e SHRIVASTAVA, S.K. ( 1982 ), "Realiable Remote Calls for Distributed UNIX: An Implementation Study", Proceedings 2nd Symp. on Reliability in Distributed Software and Database Systemas, Pittsburh, p.127-133, july 1982. ( reimpresso em SHRIVASTAVA, 1985 ).

PANZIERI, F. e SHRIVASTAVA, S.K. ( 1985 ), "Rajdoot: A Remote Procedure Call Mechanism Supporting Orfhan Detection and Killing", Technical Report Series, n.200. Computing Laboratory. University of Newcastle upon Tyne. 1985.

PANZIERI, F. e SHRIVASTAVA, S.K. ( 1988 ), "Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing ", IEEE Transactions on Software Engineering, vol.14(1), january 1988.

PETERSON, J.L. e SILBERSCHATZ, A. ( 1983 ), "Operating System Concepts", Addison-Wesley Publishing Company, Inc., 1983.

PETERSON, J.L. e SILBERSCHATZ, A. ( 1985 ), "Operating System Concepts ( second edition )", Addison-Wesley Publishing Company Inc., 1985.

POPEK, G.J. e WALKER, B.J. ( 1985 ), "The LOCUS Distributed System Architecture", The MIT Press series in computer system, Cambridge, Massachusetts, London, England, 1985.

POWELL, M.L. e MILLER, B. ( 1983 ), "Process Migration in DEMOS/MP", Proceedings of 9th Symposium of Operating Systems Principles, 110-119, october 1983.

POWELL, M.L. e PRESOTO, D.L. ( 1983 ), "Publishing: A Reliable Broadcast Communication Mechanism", Proceedings of 9th Symposium of Operating Systems Principles, 100-109, october 1983.

RANDELL, B. ( 1979 ), "System Reliability and Structuring", Capítulo 1, p.1-18, de ANDERSON e RANDELL ( 1979 ).

RANDELL, B. ( 1975 ), "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, SE-1, N.2, june 1975. ( reimpresso em SHRIVASTAVA, 1985 ).

RANDELL, B. ( 1980 ), "Reliability and Integrity of Distributed Computing Systems ", Technical Report Series, n.154, Computing Laboratory, University of Newcastle upon Tyne, England, july 1980.

RANDELL, B. ( 1984 ), "Fault Tolerance and System Structuring", Proceedings of 4th Jerusalem Conf. on Information Technology, may 1984. ( reimpresso em SHRIVASTAVA, 1985 ).

RANDELL, B. ( 1987 ), "System Design and Structuring for Dependability", Technical Report Series, n.232, Computing Laboratory, University of Newcastle upon Tyne, England, , february 1987.

SEGRE, L.M. ( 1981 ), "Mecanismos de Comunicação para Processos Concorrentes", Relatório Técnico do Programa de Engenharia de Sistemas e Computação - COOPE/UFRJ, ES 09-81.

SEGRE, L.M. e KIRNER, C. ( 1982 ), "Primitivas de Comunicação e Sincronização por Troca de Mensagens: Uma Análise", Gráfica da UFSCar - Relatório Técnico, 1982.

SEGRE, LIDIA M. e SANTOS, SUELI M. ( 1981 ), "O Conceito de Monitor como Instrumento de Sincronização em Programação Concorrente", Relatório Técnico do Programa de Engenharia de Sistemas e Computação COOPE/UFRJ, ES 03-81.

SHOOMAN, M.L. ( 1979 ), "Software Reliability", Capítulo 9, p.355-422, de ANDERSON e RANDELL (1979 ).

SHRIVASTAVA, S.K. ( 1978 ), "Sequencial Pascal with Recovery Blocks", Software-Practice and Experience, vol. 8, p. 177-185, JOHN Wiley & Sons, Ltd., 1978. ( reimpresso em SHRIVASTAVA, 1985 ).

SHRIVASTAVA, S.K. ( 1979 ), "Concurrent Pascal with Backward Error Recovery: Language Features and Examples", Software-Practice and Experience, vol. 9, p.1001-1020, 1979. ( reimpresso em SHRIVASTAVA, 1985 ).

SHRIVASTAVA, S.K. ( 1981 ), "Structuring Distributed Systems for Recoverability and Crash Resistance", IEEE Transactions on Software Engineering, SE-7, n.4, p.436-447, july 1981. ( reimpresso em SHRIVASTAVA, 1985 ).

SHRIVASTAVA, S.K. ( 1983 ), "On the Treatment of Orphans in Distributed System", Technical Report, n.180, University of Newcastle upon Tyne, Claremont Tower, England.

SHRIVASTAVA, S.K. ( 1985 ), "Reliable Computer Systems - Collected Papers of the Newcastle Reliability Project", Springer-Verlag, Berlin Heidelberg, 1985.

SHRIVASTAVA, S.K. ( 1986 ), "Replicated Distributed Processing", Technical Report Series, n.222. Computing Laboratory. University of Newcastle upon Tyne. 1986.

SHRIVASTAVA, S.K. e PANZIERI, F. ( 1981 ), "The Design of a Reliable Remote Procedure Call Mechanism", Technical Report Series, n.171, Computing Laboratory, University of Newcastle upon Tyne, England, october 1981.

SHRIVASTAVA, S.K. e PANZIERI, F. ( 1982 ), "The Design of a Reliable Remote Procedure Call Mechanism", IEEE Transactions on Computers, vol C-31, N.7, july 1982

SIEWIOREK, D.P. ( 1984 ), "Architecture of Fault-Tolerant Computers", Computer , vol 17 (8), august 1984.

SIEWIOREK, D.P. ( 1990 ), "Fault Tolerance in Commercial Computers", IEEE Computer Society, July 1990.

SILBERSCHATZ, A. ( 1981 ), "On the Synchronization Mechanism of the Ada Language", ACM - SIGPLAN Notices, 16(2), february 1981.

SILBERSCHATZ, A. e PETERSON, J.L. ( 1989 ), "Operating Systems Concepts", Addison Wesley Publishing Company, 1989.

SLOMAN, M.; KRAMER, J.; MAGEE, J.e TWIDLE, K. ( 1983 ), "A Flexible Communication System for Distributed Computer Control", Research Report Doc 83/11, Depart. of Computing, Imperial College, 180 QeensGate, London, SW7 2BZ.

SLOMAN, M.; MAGEE, J. e KRAMER, J. ( 1984 ), "Building Flexible Distributed Systems in CONIC", Proceedings SERC Distributed Computing 84 Conf., University of Sussex, Bringhton, september 1984.

SMITH, J.M. ( 1988 ), "A Survey of Process Migration Mechanism", ACM - Operating System Review, vol.22(3), july 1988.

SPECTOR, A.Z. ( 1982 ), "Performing Remote Operations Efficiently on a Local Computer Network", Communications of the ACM, vol 25 (4), april 1982.

STONE, R.F. ( 1989 ), "Realiable Computing Systema: A Review", Technical Report, University of York, Departament of Computer Science, Heslington, York, England, 1989.

SWEET, R.F. ( 1985 ), "The Mesa Programming Enviromment", ACM - SIGPLAN Notices, vol 20 (7), july 1985. Proc. ACM-SIGPLAN 85 Symp. on Lang. Issues in Prog.Env.

TANENBAUM, A.S. e MULLENDER, S.J. ( 1981 ), "An Overview of the Amoeba Distributed Operating System", Operating Systems Review, vol 15(3), 51-64, july 1981

TANENBAUM, A.S. e RENESSE, R.V. ( 1985 ), "Distributed Operating Systems", ACM Computing Surveys, vol 17 (4), december 1985.

TIZATO, F. e ZICARI, R. ( 1985 ), "Mechanism for Dinamic Configuration of a Locally Distributed System", Cnet 2nd Consiglio Nazionale Delle Ricerche Progetto Finalizzato informatica Sottoprogeto Pi, Distributed Systems on Local Network, Pisa, june 1985.

WIRTH, N. ( 1983 ), "Programming in Modula-2", Springer-Verlag, Berlin Heidelberg 1983.

WOOD, W.G. ( 1981 ), "Recovery Control of Communicating Processes in a Distributed System", IEEE Digest of Papers, p.159-164, june 1981. ( reimpresso em SHRIVASTAVA, 1985 ).

ZAYAS, E.R. ( 1987 ), "Attacking the Process Migration Botttleneck", Proceeding of the 11th Symposium on Operating Systems, Austin, Texas, 1987.