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

ADC - Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável

por

PATRÍCIA PITTHAN DE ARAÚJO BARCELOS

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

Prof. Taisy Silva Weber
Orientadora

Porto Alegre, junho de 1996.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

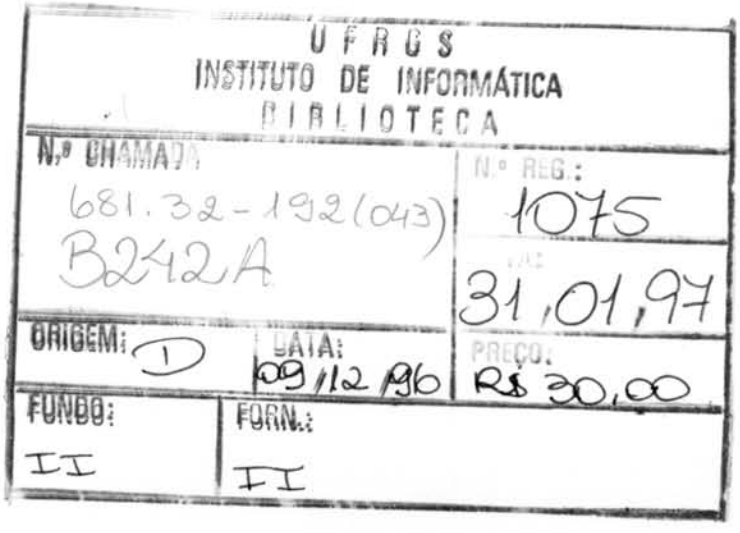
Barcelos, Patrícia Pitthan de Araújo

ADC - Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável / Patrícia Pitthan de Araújo Barcelos - Porto Alegre: CPGCC da UFRGS, 1996.

113 p.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1996. Orientadora: Weber, Taisy Silva.

1.Tolerância a Falhas. 2.Sistemas Distribuídos. 3. Comunicação de Grupo. 4.Difusão Confiável. 5.Consenso. I.Weber, Taisy Silva. II.Título.



Confiabilidade de Computadores - SBC
confiabilidade: Computadores Tolerancia. Falhas Sistemas distribuidos Difusão confiavel CNPq 1.03.03.00-6

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Hélgio Trindade
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Cláudio Scherer
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Sumário

Lista de Abreviaturas

ACK	*Acknowledgement*
ADC	Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável
AS	*Authorization Server*
BS	*Boot Server*
CPGCC	Centro de Pós-Graduação em Ciência da Computação
CPU	*Central Processing Unit*
DCL	*Distributed Computing Layer*
DSM	*Distributed Shared Memory*
FIFO	*First In First Out*
FS	*File Server*
HetNOS	*Heterogeneous Network Operating System*
NACK	*Negative Acknowledgement*
NFS	*Network File System*
NS	*Name Server*
PC	*Personal Computer*
RPC	*Remote Procedure Call*
SS	Servidores do Sistema
TCP	*Transport Control Protocol*
TLI	*Transport Layer Interface*
TS	*Type Server*
UFRGS	Universidade Frederal do Rio Grande do Sul

Lista de Figuras

Lista de Tabelas

Resumo

Uma tendência recente em sistemas de computação é distribuir a computação entre diversos processadores físicos. Isto conduz a dois tipos de sistemas: sistemas fortemente acoplados e sistemas fracamente acoplados. Este trabalho enfoca os sistemas de computação classificados como fracamente acoplados, ou sistemas distribuídos, como são popularmente conhecidos.

Um sistema distribuído, segundo [BAB 86], pode ser definido como um conjunto de processadores autônomos que não compartilham memória, não tem acesso a *clocks* globais e cuja comunicação é realizada somente por troca de mensagens.

As exigências intrínsecas de sistemas distribuídos compreendem a confiabilidade e a disponibilidade. Estas exigências têm levado a um crescente interesse em técnicas de tolerância a falhas, cujo objetivo é manter a consistência do sistema distribuído, mesmo na ocorrência de falhas.

Uma técnica de tolerância a falhas amplamente utilizada em sistemas distribuídos é a técnica de difusão confiável. A difusão confiável é uma técnica de redundância de software, onde um processador dissemina um valor para os demais processadores em um sistema distribuído, o qual está sujeito a falhas [BAB 85].

Por ser uma técnica básica de comunicação, diversos procedimentos de tolerância a falhas baseiam-se em difusão confiável. Este trabalho descreve a implementação de um ambiente de apoio a sistemas distribuídos intitulado *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*. Neste ambiente são utilizados os recursos da difusão confiável para a obtenção de uma concordância entre todos os membros do sistema livres de falha. Esta concordância, conhecida como consenso, é obtida através de algoritmos de consenso, os quais visam introduzir o grau de confiabilidade exigido pelos sistemas distribuídos.

O ADC (*Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável*) foi desenvolvido em estações de trabalho SUN (SunOS) utilizando o sistema operacional de rede heterogêneo HetNOS [BAA 93] desenvolvido na UFRGS.

O ambiente foi implementado com base em um estudo realizado sobre protocolos de difusão confiável [BAR 94]. Através da implementação do ADC é possível simular a execução de protocolos de difusão confiável aplicando modelos propostos para os mesmos. Desta execução são extraídos resultados, sobre os quais pode-se realizar uma análise. Esta análise tem sua fundamentação principalmente nos parâmetros de desempenho, confiabilidade e complexidade.

Tanto a implementação do ADC como a realização da análise do modelo proposto foram realizados tendo como suporte alguns dos protocolos de difusão confiável disponíveis na literatura.

O principal objetivo deste ambiente consiste na experimentação, ou seja, na verificação da relação teórico-prática dos sistemas distribuídos perante a utilização de uma técnica de redundância de software, a difusão confiável. Através deste ambiente torna-se possível a determinação de parâmetros tais como o número de mensagens de difusão trocadas entre os processos, o número de mensagens de retransmissão enviadas, o número de mensagens emitidas durante todo o processamento do modelo, etc. Estes parâmetros resultam numa análise consistente de protocolos de difusão confiável.

PALAVRAS-CHAVE: Tolerância a Falhas, Sistemas Distribuídos, Comunicação de Grupo, Difusão Confiável, Consenso.

TITLE: "Reliable Broadcast Protocols Experimentation and Evaluation Environment (ADC)"

Abstract

A recent trend in computing systems is to distribute the computation between several physical processors. This leads to two different systems: closely coupled systems and loosely coupled systems. This work focuses on computing systems classified as loosely coupled or distributed systems, as they are commonly known.

According to [BAB 86], a distributed system can be defined as a set of autonomous processors with no shared memory, no global clocks and whose comunication is performed only by message exchange.

The inherent requirements of distributed systems include reliability and availability. These have caused an increasing interest in fault tolerance techniques, whose goal is to keep the distributed system consistent despite failures.

A fault tolerance technique largely used in distributed systems is reliable broadcast. Reliable broadcast is a software redundancy technique, where a processor disseminates a value to other processors in a distributed system, in which failures can occur [BAB85].

Because it is a basic communication technique, several fault tolerance procedures are based on reliable broadcast. This work describes the implementation of a support environment for distributed systems called *Reliable Broadcast Protocols Experimentation and Evaluation Environment (ADC)*. Reliable broadcast resources are used in this environment to obtain an agreement among all off-failure system components. This agreement, called consensus, has been obtained through consensus algorithms, which aim to introduce the reliability degree required in distributed systems.

The ADC has been developed in Sun workstation (SunOS) using the heterogeneous operating system HetNOS [BAA 93] which was developed at UFRGS.

The environment has been implemented based on a research about reliable broadcast protocols [BAR 94]. Through the ADC it is possible to simulate the execution of reliable broadcast protocols applying proposed models to them. From this execution results are extracted, and over them analysis can be done. This analysis has been based essentialy in parameters such as performance, reliability and complexity.

Some classical reliable broadcast protocols were used as a support to ADC implementation and model analysis.

The main goal of this environment consists in validating diffusion protocols in a practical distributed systems environment, facing reliable broadcast. Through this environment it can be possible the analysis of important parameters resolution such as the number of messages exchanged between process, the number of retransmission of messages sent, the number of messages sent during the whole model processing, others. These parameters result in a consistent analysis of reliable broadcast protocols.

KEY WORDS: Fault Tolerance, Distributed Systems, Group Communication, Reliable Broadcast, Consensus.

1 Introdução

Durante a década passada, verificou-se o surgimento de um novo ambiente de processamento, o ambiente distribuído. Tal fato se deveu, principalmente, aos avanços em *hardware*, os quais reduziram substancialmente o custo dos processadores, popularizando assim os computadores pessoais e, mais tarde, as estações de trabalho. O *hardware* de comunicação também evoluiu, o que resultou no desenvolvimento das redes locais de alto desempenho. Com isso, criou-se o potencial para compartilhamento de recursos como processadores, dispositivos de impressão e componentes secundários de armazenamento.

Porém, o *software* não conseguiu acompanhar o grande desenvolvimento apresentado pelo *hardware*, cujo avanço criou uma gama de possibilidades no que se refere ao processamento distribuído. A demanda por desenvolvimento a nível de *software* distribuído resultou no fortalecimento da pesquisa nessa área.

Uma tendência recente em sistemas de computação é distribuir a computação entre vários processadores físicos. Há basicamente dois esquemas para construção de tais sistemas [SIL 91]: os sistemas fortemente acoplados e os sistemas fracamente acoplados. Em um sistema fortemente acoplado, a comunicação usualmente é realizada por intermédio da memória compartilhada. Já em um sistema fracamente acoplado, os processadores não compartilham memória ou *clock*. Ao invés disso, cada processador tem sua própria memória local. Os processadores comunicam-se um com o outro através da rede de comunicação. Tais sistemas são chamados de sistemas distribuídos [SIL 91].

A utilização crescente de estações de trabalho de alto desempenho interligadas por redes de comunicação de alta velocidade tem gerado um interesse cada vez maior por sistemas distribuídos. A redundância implícita a um sistema distribuído oferece novas e interessantes oportunidades, antes ignoradas ou desconsideradas em sistemas centralizados por questões estruturais ou de viabilidade.

Até hoje não há unanimidade no conceito de *sistemas distribuídos*, mas pode-se definir como um conjunto de processadores autônomos que não compartilham memória, não tem acesso a *clocks* globais e comunicam-se somente por troca de mensagens [BAB 86].

Em sistemas centralizados monoprocessados tradicionais, o processador é um recurso de cujo funcionamento depende vitalmente o restante do sistema. Em sistemas distribuídos, a profusão de elementos processadores possibilita a distribuição uniforme de tarefas entre os mesmos, com a finalidade de aumentar o desempenho global do sistema [BEL 92]. Também torna-se possível reiniciar computações localizadas em processadores que venham a falhar. Comparativamente a um sistema centralizado

tradicional, um sistema distribuído apresenta vantagens, tais como a possibilidade de expansão gradual da capacidade computacional total do sistema e a multiplicidade de elementos processadores.

A segurança de funcionamento de um sistema, também denominada dependabilidade, é definida como a qualidade do serviço oferecido pelo mesmo conforme percebido pelos seus usuário [LAP 85]. Para mensurar tal nível de segurança, empregam-se duas medidas básicas: confiabilidade e disponibilidade.

Confiabilidade é uma medida de sucesso na realização de um serviço, em um intervalo de tempo determinado. É também expressa como a probabilidade mínima aceitável de o sistema funcionar corretamente nesse intervalo. O usuário percebe dois estados fornecidos pelo sistema: serviço correto, quando o serviço executado corresponde à especificação; e serviço incorreto, quando o serviço executado difere da especificação proposta para o mesmo. A transição entre esses dois estados se dá através da ocorrência de dois eventos: o surgimento de uma falha e a correspondente operação de reparação. A confiabilidade é inversamente proporcional à freqüência de tais eventos [JAL 94].

A disponibilidade é definida como o intervalo de tempo no qual o sistema satisfaz a especificação, ou seja, é a probabilidade de o sistema fornecer o serviço corretamente. Usualmente, a melhoria da confiabilidade implica um aumento da disponibilidade de um sistema. Cabe ressaltar, entretanto, que, eventualmente é possível que essas medidas sejam discrepantes. Um sistema que falha frequentemente, porém tem uma reparação rápida, apresenta uma confiabilidade baixa e uma disponibilidade alta. Inversamente, um sistema que fornece serviço correto por um período de tempo curto, possui alta confiabilidade e baixa disponibilidade. A adequação de tais medidas é função de cada aplicação. Aplicações que impliquem riscos de vida normalmente requerem uma confiabilidade alta, ao passo que em aplicações não críticas uma alta disponibilidade é mais importante. Exemplo clássicos destas aplicações são, respectivamente, o computador de bordo de um avião e uma central telefônica [JAL 94].

Os sistemas distribuídos apresentam como característica básica as exigências de confiabilidade e disponibilidade. Tais exigências têm levado a um crescente interesse em técnicas de tolerância a falhas, cujo objetivo é manter a consistência do sistema distribuído, mesmo na ocorrência de falhas. Para tanto, utiliza-se o recurso de mascaramento de falhas, o qual baseia-se no fato de que as falhas não devem ser refletidas no comportamento externo do sistema [WEB 90].

As técnicas de tolerância a falhas caracterizam-se pela redundância, ou seja, a presença de componentes ou informações replicadas [CRI 91]. Este trabalho descreve um ambiente, intitulado *Ambiente para Experimentação e Avaliação de Protocolos de*

Difusão Confiável (ADC), cujo objetivo é a análise de uma técnica de redundância de *software* [VER 89]: a difusão confiável.

Na técnica de difusão confiável, ou disseminação confiável, um processador dissemina um determinado valor para todos os outros processadores num sistema distribuído, onde ambos, processadores e componentes de comunicação, estão sujeitos a falhas [BAB 85]. O interesse por difusão confiável surgiu tanto com a necessidade de obtenção de um meio de otimização da comunicação entre os diversos processadores que compõem um sistema distribuído, como pela exigência de consenso entre os mesmos.

Difusão confiável é uma técnica básica de comunicação, sobre a qual diversos procedimentos de tolerância a falhas se baseiam. Exemplos de aplicações de difusão confiável são: o diagnóstico de falhas em sistemas distribuídos, a localização de objetos em um serviço distribuído, tal como um arquivo em um serviço de arquivos distribuído, e a replicação de dados em sistemas distribuídos, a qual é bastante utilizada em atualizações múltiplas [COU 94].

Existem, na literatura, diversos protocolos que implementam difusão confiável em sistemas distribuídos [BAB 85], [BIR 87], [CHA 84], [CRI 85], [JAL 94], etc. Tais protocolos apresentam como características básicas: a existência ou não de limitação no tempo de entrega das mensagens, o tipo de ordenação de mensagens implementado, os tipos de falhas suportadas e o tipo de acordo para a determinação da concordância entre os membros ativos do sistema. Através destas características, tornou-se possível a implementação do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*, visando a obtenção de um ambiente do qual pudessem ser extraídos resultados por meio de análise de protocolos existentes.

O ADC (*Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável*) foi desenvolvido em estações de trabalho SUN (SunOS) utilizando o sistema operacional de rede heterogêneo HetNOS. O sistema HetNOS, igualmente, foi desenvolvido como dissertação de mestrado no CPGCC da UFRGS.

O ambiente foi implementado com base em um estudo realizado sobre protocolos de difusão confiável [BAR 94]. Através da implementação do ADC é possível simular a execução de protocolos de difusão confiável por meio de modelos propostos para os mesmos. Da execução destes modelos são extraídos resultados, sobre os quais pode-se realizar uma análise. Esta análise tem fundamentação principalmente nos parâmetros de desempenho, confiabilidade, complexidade e custo. Em [BAR 95] foram apresentados alguns resultados obtidos durante a fase de desenvolvimento do ambiente.

Tanto a implementação do ADC como a realização da análise do modelo proposto foram realizados tendo como suporte alguns dos protocolos de difusão confiável disponíveis na literatura.

O objeto central de interesse do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)* é a implementação de difusão confiável em sistemas distribuídos, visando disponibilidade e confiabilidade, sendo esta última obtida por meio de algoritmos de consenso. Através deste ambiente de experimentação é possível desenvolver um processo de análise qualitativa e validação dos resultados, viabilizando sua aplicação em sistemas distribuídos. Este trabalho encontra-se organizado da seguinte maneira.

No capítulo 2 são apresentados os conceitos básicos a respeito de sistemas distribuídos. Alguns destes conceitos são relativos à sincronização e ordenação de eventos em sistemas distribuídos, onde são analisados os tipos de ordenação de mensagens existentes. Aspectos de comunicação e *bufferização* de mensagens também são observados neste capítulo.

O capítulo 3 enfoca difusão confiável apresentando uma classificação de falhas, a qual foi adotada na implementação do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*. A partir daí, são apresentadas as propriedades da difusão confiável, bem como sua classificação. São estudados ainda alguns protocolos de comunicação conhecidos na literatura, onde se inclui o Problema dos Generais Bizantinos, que constitui um bloco básico sobre o qual baseiam-se diversos protocolos de comunicação.

No capítulo 4 é apresentada a plataforma de desenvolvimento do ADC, ou seja, é descrito o sistema sobre o qual o ADC foi implementado. O ADC foi desenvolvido sobre o sistema operacional de rede heterogêneo HetNOS. A descrição apresentada neste capítulo envolve aspectos da estrutura física do HetNOS, sua arquitetura sobre o *Unix*, bem como as primitivas de comunicação, sincronização e gerência de processos mais utilizadas pelo ADC. Este capítulo aborda ainda uma discussão sobre a opção pelo HetNOS como plataforma de desenvolvimento do ADC.

O capítulo 5 descreve o *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*. Aspectos relacionados à implementação e à estrutura do ambiente são apresentados neste capítulo. Ainda no capítulo 5 são expostos detalhadamente os seis módulos componentes do ADC, assim como os relacionamentos entre os mesmos quando da execução da experimentação e análise.

No capítulo 6 é apresentada a descrição do funcionamento do ADC como um todo. Isto é, a partir das funções de cada módulo, descritas no capítulo 5, organizou-se uma descrição sequencial do procedimento do ADC. Da mesma forma que o capítulo anterior, aspectos de implementação são apresentados, bem como dificuldades e

restrições impostas pelo sistema no qual o ADC foi desenvolvido, o HetNOS. O capítulo 6 exibe ainda uma descrição da simulação da execução de um modelo. São expostas as etapas de experimentação e análise, bem como os resultados intermediários da execução destas duas etapas.

O capítulo 7 apresenta um relato sobre as dificuldades encontradas no desenvolvimento do ambiente, desde a fase de projeto do mesmo até a sua implementação. São abordadas ainda algumas conclusões as quais se chegou com a implementação do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*.

2 Sistemas Distribuídos

Um sistema distribuído consiste de muitos nodos, cada um destes contendo um processador, geograficamente distantes, porém conectados por uma rede de comunicação. Os nodos de um sistema distribuído são ditos autônomos por serem classificados como fracamente acoplados, e por comunicarem-se exclusivamente através de troca de mensagens. Outra característica peculiar destes sistemas é a ausência tanto de memória compartilhada, como de *clocks* globais [BIR 87]. Do ponto de vista de um processador específico, em um sistema distribuído, o restante dos processadores, bem como seus respectivos recursos são remotos, enquanto seus próprios recursos são considerados recursos locais [SIL 91].

Os processadores em um sistema distribuído podem variar tanto em tamanho como em função. Eles podem incluir pequenos microprocessadores, estações de trabalho, minicomputadores e vários sistemas de computadores de propósito geral. Estes processadores são referenciados na literatura por diferentes nomes, tais como *sites*, nodos, computadores e assim por diante, dependendo do contexto no qual os mesmos são mencionados [SIL 91].

Há uma variedade de razões para a construção de sistemas distribuídos, algumas destas são [SIL 91]:

- *compartilhamento de recursos*: se um número de diferentes nodos (com diferentes capacidades) são conectados um ao outro, então um usuário em um nodo pode ser capaz de utilizar os recursos disponíveis em outro. Em geral, compartilhamento de recursos em um sistema distribuído oferece mecanismos para compartilhamento de arquivos, processamento de informação em um sistema de banco de dados distribuído, impressão remota de arquivo, uso de periféricos de *hardware* especializados remotamente localizados, etc;
- *velocidade de computação*: se uma determinada computação pode ser particionada em um determinado número de subcomputações, as quais podem rodar concorrentemente, então um sistema distribuído pode permitir a distribuição da computação entre os vários nodos que o compõe. Desta forma, tem-se um aumento na velocidade de computação introduzido pela concorrência. Similarmente, se um determinado nodo está sobrecarregado de tarefas, algumas delas podem ser movidas para outros nodos levemente carregados. Este movimento de tarefas é denominado *balanceamento de carga*;
- *confiabilidade*: se um nodo falha em sistema distribuído, os demais nodos podem continuar a operação normal. Se o nodo é composto por um número de processadores autônomos, a falha de um deles poderia não afetar o sistema

como um todo. Se, por outro lado, o sistema é composto por um número de processadores, cada um dos quais responsável por alguma função específica do sistema, então uma única falha pode efetivamente parar a operação de todo o sistema. Em geral, se existe redundância suficiente no sistema, ou seja, tanto no que diz respeito ao *hardware*, como aos dados, o sistema pode continuar com sua operação, mesmo se alguns de seus nodos tiverem falhado;

- *comunicação*: há muitas instâncias nas quais os processadores necessitam trocar dados um com o outro em um sistema distribuído. Quando alguns nodos estão conectados um ao outro por uma rede de comunicação, os processadores em diferentes nodos tem a oportunidade de trocar informação. Usuários podem iniciar transferências de arquivos ou se comunicar um com o outro através de correio eletrônico, por exemplo.

Um modelo para sistema distribuído típico pode ser composto por diversas unidades interconectadas por uma rede de comunicação. Uma unidade compreende todo o *hardware* incluindo a interface de rede, e o sistema operacional. Em alguns casos é interessante ver a unidade como um processador mais a(s) interface(s) de rede. Cada processador pode ter um número arbitrário de processos do usuário executando nele. A única forma de comunicação entre as unidades é através da rede de comunicação, em função da ausência de áreas de memória compartilhadas. As interfaces e a rede de comunicação compõem o que costuma ser chamado de *sistema de comunicação* [TAN 89].

Há duas características importantes em um sistema de comunicação com relação à tolerância a falhas [MÓR 94]:

- *robustez*: corresponde à redundância inerente da interconexão oferecida. Representa a habilidade do subsistema de comunicação em manter um fluxo de informação na presença de falhas de barramentos, *links*, roteadores, etc;
- *reconfigurabilidade*: indica a facilidade de utilização de tal redundância na prática. Esta medida é relevante em função da sobrecarga que pode surgir devido à aplicação de algoritmos de roteamento quando da ocorrência de problemas de comunicação.

Segundo [JAL 94], um sistema distribuído pode ser visto de duas maneiras: como definido pelos componentes físicos do sistema (modelo físico), e como definido do ponto de vista do processamento (modelo lógico). Ambos os modelos são de fundamental importância sob o aspecto de tolerância a falhas.

O ponto de vista do processamento (modelo lógico) é importante por ser a maneira pela qual o usuário enxerga o sistema. Os serviços definidos neste modelo são aqueles para os quais a confiabilidade é desejada. O modelo físico é também importante porque o processamento é realizado na rede física, e os componentes desta rede física são os que falham. Neste contexto, o objetivo de tolerância a falhas em sistemas

distribuídos é assegurar que alguma propriedade ou serviço no modelo lógico seja preservada apesar da ocorrência de falhas em alguns componentes do sistema físico [JAL 94].

Os principais componentes de um sistema distribuído são o processador, a rede de comunicação, os *clocks* e o *software*. Em sistemas distribuídos, estes componentes são considerados atômicos. É a falha destes componentes que o esquema de tolerância a falhas visa mascarar, tal que o sistema como um todo não falhe. Frequentemente, um sistema distribuído é modelado como tendo nodos e uma rede de comunicação como componentes básicos, sem considerar a estrutura interna dos nodos. Neste modelo, a falha de um nodo e a falha da rede de comunicação são os principais componentes de falha [JAL 94].

Em sistemas ponto-a-ponto, a rede de comunicação consiste de um conjunto de *links*, cada *link* conectando dois nodos através de suas respectivas interfaces de rede. Neste modelo, um sistema distribuído pode ser representado como um grafo, onde os nodos no grafo representam nodos do sistema, e as flechas no grafo representam os *links* de comunicação do sistema. A maneira na qual os diferentes *links* são conectados a diferentes nodos é chamada topologia da rede. Em um sistema distribuído é de extrema importância o modo pelo qual os nodos estão ligados entre si, uma vez que a topologia da rede exerce influência direta no desempenho da mesma [TAN 89].

O envio de mensagens em redes ponto-a-ponto exige protocolos de comunicação. Protocolos são necessários devido a natureza autônoma de seus nodos e a separação geográfica entre eles. Tipicamente, uma parte do protocolo é executada pelo *hardware* na interface de rede, e o restante é realizado pelo *software* do sistema [JAL 94].

Um sistema distribuído pode ser projetado para ser tolerante a falhas de duas maneiras [SIN 94]:

- através do mascaramento de falhas;
- através da exibição de um comportamento bem definido no evento da falha.

Quando um sistema é projetado para mascarar falhas, ele continua a realizar a sua função especificada no evento de uma falha. Um sistema projetado para exibir um comportamento bem definido pode ou não realizar a função especificada no evento de uma falha. Contudo, o sistema pode facilitar ações apropriadas para recuperação. Um exemplo de um comportamento bem definido durante uma falha são as mudanças realizadas em um banco de dados por uma transação. Estas somente são visíveis para outras transações se a transação obteve *commit* com sucesso. Caso a transação falhe, as mudanças realizadas no banco de dados pela transação falha não são visíveis à outras transações, não afetando, com isso, as demais transações [SIN 94].

Uma abordagem chave utilizada para tolerar falhas é a técnica de redundância. A técnica de redundância compreende uma técnica de mascaramento de falhas [MÓR 94]. Nesta abordagem, um sistema pode usar um número múltiplo de processos, um número múltiplo de componentes de *hardware*, múltiplas cópias de dados, e assim por diante, cada um com sua semântica de falhas independente. Com isso, a falha de um componente não afeta a operação dos demais componentes do sistema.

O crescente interesse por sistemas distribuídos tem contribuído para a utilização de técnicas de tolerância a falhas, com a finalidade de garantir confiabilidade e disponibilidade ao sistema. Contudo, a distribuição e a replicação de componentes implicam num custo adicional de gerência, o que pode comprometer o desempenho do sistema como um todo. Neste contexto, difusão confiável aparece como uma alternativa atraente mesmo para aplicações em tempo real [DUE 92].

Na presença de falhas, classificadas na seção 3.1, a difusão torna-se uma tarefa complexa, pois ela precisa tratar a possibilidade de uma parte dos processos participantes da difusão falhar durante a mesma. A falha do transmissor ou uma falha de mensagem pode fazer com que uma mensagem seja entregue para alguns e não para todos os processos destino [DUE 92]. Para garantir a validade e correção dos dados transferidos entre as unidades deve-se manter a continuidade da comunicação assegurando a consistência dos dados. Tal garantia é especialmente importante em operações do tipo *broadcasting* (comunicação um-para-todos [1:N], onde N é o número de unidades do sistema) e *multicasting* (comunicação um-para-alguns [1:M], onde M<N), onde todos os receptores obrigatoriamente necessitam receber a mesma informação do emissor [COU 94]. O problema de alcançar consenso através de *broadcasting* em um sistema distribuído sujeito a falhas arbitrárias é conhecido como Problema dos Generais Bizantinos [LAM 82]. Este problema é visto na seção 3.4.1.

O tipo de comunicação de grupo enfocado neste trabalho é *broadcasting*. Tal comunicação é implementada supondo uma rede do tipo ponto-a-ponto. Neste contexto, como base para o desenvolvimento do ambiente, foram estudados alguns dos protocolos de comunicação existentes na literatura. Com base neste estudo, em [BAR 94] foi apresentada uma análise de protocolos de comunicação pertencentes às classes síncrona e assíncrona. A importância da classificação dos protocolos segundo a propriedade de terminação (síncrona e assíncrona), enfatizada em [BAR 94], reside no fato de que a existência de limitação na realização de uma determinada tarefa em sistemas distribuídos é frequentemente necessária.

Um sistema é considerado síncrono se, e somente se, consegue executar a seqüência de instruções a ele destinadas dentro de um limite de tempo finito e conhecido [CRI 91]. A principal vantagem de um sistema síncrono, sob o enfoque de tolerância a falhas, é que a falha de um componente do sistema pode ser deduzida pelos demais devido a ausência de resposta dentro de um intervalo de tempo definido [CRI 91]. Nessas situações, pode ser usado o recurso de *timeout* para detecção de falhas ou

mensagens perdidas. [LAM 84]. Os sistemas assíncronos, por sua vez, podem ser deduzidos estar em *deadlock* por não apresentarem a característica de terminação finita.

2.1 Sincronização e Ordenação de Eventos

Em um sistema centralizado é sempre possível determinar a ordem na qual dois eventos ocorreram, desde que haja uma única memória e *clocks* comuns. Em muitas aplicações, é de extrema importância ser capaz de determinar em que ordem os eventos ocorreram. Por exemplo, em um esquema de alocação de recursos, somente pode-se especificar que o recurso pode ser usado após o mesmo ter sido concedido. Em um sistema distribuído, no entanto, não há memória e *clock* comuns. Portanto, é impossível, algumas vezes, determinar a ordem de ocorrência dos eventos [SIL 91].

A comunicação entre processos em um sistema distribuído geralmente exige algum tipo de sincronização. Esta sincronização também pode ser vista como uma espécie de obrigação de ordenação de eventos no sistema distribuído visando controlar a comunicação entre os processos [JAL 94]. A exigência de sincronização entre os processos de um sistema distribuído se deve principalmente a ausência de *clocks* globais [LAM 85].

Para obter um comportamento globalmente correto com relação às aplicações de grupo, é necessário sincronizar a ordem na qual as ações são realizadas, principalmente quando membros do grupo agem de forma independente [BIR 93].

Os sistemas distribuídos considerados neste trabalho apresentam três tipos de ordenação de eventos: ordenação causal, ordenação total e ordenação FIFO. Há ainda aqueles sistemas nos quais a ordenação se faz desnecessária, fazendo parte dos eventos sem ordenação.

A ordenação visa assegurar que todos os membros do sistema apresentem o mesmo conjunto de mensagens numa mesma ordem, possibilitando assim, uma visão única aos participantes sobre o conjunto de mensagens difundido [CHA 84].

A ordenação dos eventos pertencentes ao mesmo nodo se dá através do *clock* local. O problema surge com a necessidade de ordenação de eventos pertencentes a diferentes nodos, uma vez que os tempos dos diferentes eventos são medidos por diferentes *clocks*. Para resolver questões deste tipo, assume-se os nodos de um sistema distribuído apresentam tanto *clocks* físicos como *clocks* lógicos [JAL 94]. Segundo [COU 94], esta é a suposição utilizada para aproximar a sincronização de *clocks* físicos, visto que a introdução de *clocks* lógicos define uma ordem nos eventos sem medir o tempo físico no qual os mesmos ocorreram.

Um *clock* em um sistema distribuído possui as seguintes finalidades [DRU 93]:

- definir uma ordenação de eventos que preservam relações de causa e efeito;
- coordenar processadores para que eles possam realizar ações aproximadamente no mesmo tempo real;
- permitir a dedução sobre o estado global do sistema somente com base em informações locais.

Um *clock* lógico representa uma maneira de fixar um número a um evento, este número identifica o tempo de ocorrência do evento. Em um sistema de *clocks*, cada nodo tem seu *clock* local, tal que os tempos dos eventos sejam consistentes com a relação de ordenação [LAM 78]. Fazer uso de um *clock* lógico significa fixar um *timestamp* (carimbo de tempo) ao evento a fim de que a ordenação seja preservada. Os *clocks* lógicos não têm relação com os *clocks* privados dos nodos e seus tempos não têm relação com o tempo físico. Os *clocks* lógicos podem ser usados tanto para satisfazer a condição de ordenação total, como parcial dos eventos de um sistema distribuído [SRI 87].

Timestamps são valores de dados armazenados, os quais representam o tempo no qual algum evento ocorreu. Em sistemas distribuídos, um *timestamp* gerado em um nodo pode ser passado através de mensagens para outros nodos, a fim de que eles possam ser comparados com os *timestamps* gerados localmente [COU 94].

Do ponto de vista de qualquer processo, os eventos são ordenados unicamente pelos tempos mostrados em seus *clocks* locais. Segundo [LAM 78], enquanto não é possível a existência de *clocks* perfeitamente sincronizados em um sistema distribuído, não se pode usar o tempo físico para determinar a ordem de qualquer par de eventos.

Para comunicação e sincronização algumas primitivas são necessárias. Em um sistema distribuído, onde não há memória compartilhada, sincronização e comunicação são alcançadas por primitivas de troca de mensagens [JAL 94]. As primitivas mais comuns para troca de mensagens são as primitivas *send* e *receive*, para envio e recepção de mensagens, respectivamente. De acordo com as características do sistema distribuído, estas primitivas podem ser *síncronas* ou *assíncronas*, *bufferizadas* ou *não-bufferizadas*, e *bloqueantes* ou *não-bloqueantes*, *confiáveis* ou *não-confiáveis* [BAA 95a] [COU 94].

2.1.1 Tipos de Ordenação de Mensagens

Em sistemas distribuídos, na maioria das aplicações, é importante a determinação de qual evento ocorreu antes de outro. Tal determinação é conseguida por meio de algoritmos de ordenação de mensagens [COU 94].

A implementação destes algoritmos em sistemas distribuídos dá origem a quatro tipos de algoritmos de ordenação de mensagens, são eles: algoritmos sem ordenação,

algoritmos com ordenação FIFO, algoritmos com ordenação total e algoritmos com ordenação causal [JAL 94]. Deste quatro tipos de ordenação de mensagens surge a classificação de difusão adotada pelo *ADC* (*Ambiente para Experimentação e Avaliação de Difusão Confiável*) [JAL94], a qual é apresentada na seção 3.3. Esta classificação abrange difusão não-ordenada, difusão FIFO, difusão total e difusão causal.

2.1.1.1 Algoritmos sem ordenação

Segundo [COU 94], uma difusão de mensagens pode não ter obrigação de ordenação. Difusões que não se preocupam com ordenação de mensagens são implementadas de forma simples, uma vez que as mensagens são enviadas e recebidas sem que haja qualquer preocupação com seu sequenciamento. Tais difusões não apresentam nenhuma garantia de ordem na entrega das mensagens. Neste caso, diz-se que a difusão faz parte da classe de difusões sem ordenação.

2.1.1.2 Algoritmos com ordenação FIFO

Os algoritmos que obedecem a ordenação FIFO são aqueles em que as mensagens são entregues seguindo a ordem na qual foram enviadas pelo emissor. A implementação deste algoritmo se dá através da fixação de um número de seqüência a toda mensagem enviada [TAN 89]. Por meio deste número de seqüência, também conhecido como *timestamp*, é possível estabelecer uma ordem de difusão de mensagens, a qual deve ser obedecida por todos os nodos do sistema distribuído.

2.1.1.3 Algoritmos com ordenação total

Este tipo de ordenação garante que todas as mensagens são entregues na mesma ordem para todos os nodos do sistema, mesmo se elas forem enviadas por diferentes nodos [JAL 94]. Um protocolo que obedece ordenação total incorpora a classe de protocolo que garante a propriedade de atomicidade [CHA 84]. Os algoritmos com ordenação total apresentam um custo muito elevado devido à exigência de que todas as mensagens de difusão alcancem todos os membros do sistema na mesma ordem.

2.1.1.4 Algoritmos com ordenação causal

Algoritmos com ordenação casual baseiam-se na idéia de causalidade potencial introduzida por [LAM 78]. A implementação de algoritmos com ordenação causal supõe a existência de uma relação de causalidade potencial entre os eventos [COU 94]. Esta relação, também conhecida como aconteceu antes ("*happened before*"), descrita na

seção 3.3.5, estabelece a ordenação de eventos que apresentam uma relação de causa e efeito. Os eventos que não participam da relação de causalidade não necessitam de qualquer tipo de ordenação [COU 94].

2.2 Comunicação e *Bufferização*

Em sistemas distribuídos, onde não há memória compartilhada, a troca de mensagens é usada tanto para comunicação como para sincronização. Comunicação é alcançada por um processo que envia algum valor para outro processo, o qual recebe aquele valor. Sincronização é alcançada, uma vez que a troca de mensagem implica que o recebimento da mensagem é feito depois do seu envio [JAL 94].

Existem duas primitivas principais para troca de mensagens [BAA 95a]:

- `send(id_destino, msg)`: envia uma mensagem `msg` a um processo destino identificado por `id_destino`.
- `receive (id_origem, msg)`: requisita a recepção de uma mensagem `msg` que tenha sido enviada pelo processo `id_origem`.

A terminologia que classifica mensagens não é unânime. Pode-se dizer que existem mensagens síncronas ou assíncronas, *bufferizadas* ou *não-bufferizadas*, bloqueantes ou não-bloqueantes, confiáveis ou não-confiáveis [BAA 95a] [COU 94].

Com troca de mensagens síncrona, o emissor da mensagem fica bloqueado até a leitura da mesma pelo receptor. Assim sendo, emissor e receptor não apenas trocam dados, como também são sincronizados através de mensagens. Na recepção síncrona, o receptor espera bloqueado pela chegada de uma mensagem. Sendo assim, ambas as operações *send* e *receive* são do tipo bloqueantes [COU 94].

Com mensagens assíncronas, o emissor atua como não-bloqueante, ou seja, ele envia uma mensagem e não precisa esperar que a mensagem seja lida pelo processo receptor. Neste caso o emissor está liberado para prosseguir. Na recepção assíncrona, o processo requisita a recepção da mensagem, indicando a posição de memória para onde a mensagem deve ser copiada. O controle é retornado ao processo receptor mesmo que uma mensagem não esteja disponível, e o pedido de recepção fica pendente. Assim, há várias formas do processo saber que uma mensagem chegou:

- sendo avisado por uma interrupção gerada pelo sistema operacional (modelo assíncrono) [BAA 95a];
- fazendo com que o processo receptor verifique periodicamente uma variável que indica se a mensagem requisitada foi recebida ou não (modelo com *pooling*) [BAA 95a];

- exigindo um *timeout* para que seja especificado o intervalo de tempo após o qual a operação deve ser cancelada [COU 94] (modelo síncrono).

A troca de mensagem exige algum *buffer* entre o emissor e o receptor. O emissor coloca a mensagem no *buffer*, de onde o receptor a retira. Se a troca de mensagens é assíncrona, supõe-se que há um *buffer* infinito para armazenar mensagens. Em outras palavras, com troca de mensagens assíncrona, o emissor pode continuar a enviar mensagens as quais serão armazenadas em um *buffer* para o processo receptor consumir. Neste caso, o emissor nunca fica bloqueado, isto é, um comando *send* sempre tem êxito e um emissor pode estar arbitrariamente à frente do receptor. Contudo, o processo receptor não é não-bloqueante, uma vez que será bloqueado se não há mensagens no *buffer* esperando por ele [JAL 94].

Mensagens podem ser armazenadas tanto no nodo origem como no destino, sendo que neste último é o procedimento mais comum. Porém, em ambos os casos é necessário considerar a hipótese de limitação no espaço dos *buffers*. Portanto, é necessário decidir se o processo origem deve ser trancado ou o envio da mensagem deve falhar, no caso de estouro da capacidade do *buffer* [BAA 95a]. Contudo, em muitas situações, por propósitos práticos, o *buffer* pode ser considerado não limitado e a troca de mensagem pode ser considerada assíncrona [COU 94].

No modelo síncrono, mensagens são *não-bufferizadas*, podendo haver uma única mensagem pendente por processo. O modelo síncrono é mais simples por não haver a necessidade de gerenciar *buffers*. Por outro lado, é menos flexível, pois o nodo origem deve sempre esperar pela aceitação da mensagem, mesmo que o receptor não tenha que retornar dados, diminuindo o grau de paralelismo. Com o modelo síncrono, toda execução de um comando de comunicação representa um ponto de sincronização onde ambos emissor e receptor sincronizam [BAA 95a].

Mensagens podem ser bloqueantes ou não-bloqueantes. No primeiro caso, o nodo origem só pode ser liberado quando um aviso do nodo destino chegar. Isto indica que a mensagem foi "seguramente" armazenada, ou seja, entregue ao processo destino. No caso não-bloqueante, o processo origem da mensagem é liberado o mais cedo possível, isto é, assim que a mensagem é copiada para o sistema. Usualmente, esta primitiva é não-confiável, na medida que o emissor não tem como saber se a mensagem foi entregue ao destino. Quanto à recepção, o controle retorna imediatamente ao processo requisitor, recebendo uma mensagem de erro como resposta [BAA 95a].

Por fim, mensagens podem ser ponto-a-ponto ou multiponto. Este modelo de mensagens existe porque muitas redes locais utilizadas para a implementação de sistemas distribuídos fornecem um modelo diferente de transmissão de mensagens de 1 origem para um conjunto **N** de processadores. Enviar uma mensagem a todos os processos do sistema distribuído significa fazer um *broadcast* de uma mensagem, enquanto em uma operação de *multicast* uma mensagem é enviada apenas a um

subconjunto de processos. Operações de *broadcast* e *multicast* não devem degradar significativamente o desempenho de uma operação normal de envio de mensagem [BAA 95a].

Um dos problemas dos modelos cuja transmissão de mensagens se dá de 1 origem para um conjunto N de processadores é a falta de confiabilidade. Isto porque não é possível garantir a entrega de todas as mensagens a seus destinos. Por outro lado, esses modelos de difusão de mensagens são bastante úteis em sistemas operacionais e sistemas *run-time* de linguagens. Tais modelos podem ser utilizados, por exemplo, para localizar uma determinada informação que está em algum dos processadores, mas não se sabe qual. Apenas o processador contendo a informação responde ao requisitor [BAA 95a].

Há uma diferença entre troca de mensagem síncrona e assíncrona e sistemas distribuídos síncronos e assíncronos. O primeiro se refere à primitivas de comunicação e o tamanho do *buffer* entre o emissor e o receptor, enquanto o segundo trata de limites em atrasos de mensagens. Em um sistema síncrono, ambas troca de mensagem síncrona e assíncrona podem ser suportadas [JAL 94].

3 Difusão Confiável

Em muitas aplicações, comunicação ponto-a-ponto é suficiente, entretanto, há aplicações onde um nodo precisa enviar uma mensagem para muitos outros nodos de um sistema distribuído. Em tais aplicações, a forma de comunicação 1-para-muitos apresenta mais vantagens. A comunicação 1-para-muitos pode se apresentar de duas maneiras: *broadcast* e *multicast* [COU 94]. A comunicação 1-para-muitos pode também ser implementada sobre redes ponto-a-ponto [JAL 94]. Baseado nesta suposição, a implementação do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)* se dá sobre redes de comunicação ponto-a-ponto.

Broadcast, cuja tradução significa difusão, é a primitiva de comunicação onde o emissor envia uma mensagem para todos os nodos do sistema distribuído. Em *multicast*, o emissor envia a mensagem apenas para um subconjunto dos nodos que compõem o sistema distribuído [JAL 94]. Este trabalho se restringe à primitiva *broadcast* para a comunicação entre os diversos nodos do sistema distribuído.

A busca por concordância entre os membros do sistema livres de falhas, mais conhecida como consenso [FIS 85], é uma característica importante sob o ponto de vista de confiabilidade em sistemas distribuídos. Neste sentido, exige-se que todos os membros do grupo tomem a mesma decisão sobre o valor difundido no sistema [JAL 94]. Este problema pode se tornar complexo na existência de nodos cujo comportamento é arbitrário. Neste caso, o protocolo de difusão deve ser robusto o suficiente para mascarar tal situação. Na seção 3.4 são discutidos aspectos relacionados ao consenso em sistemas distribuídos.

O problema de concordância em um sistema onde os componentes podem falhar de maneira arbitrária é descrito em [LAM 82] e é conhecido como Problema dos Generais Bizantinos. Este problema é apresentado na seção 3.4.1. Os protocolos utilizados para alcançar concordância em sistemas sujeitos a falhas bizantinas são chamados de Protocolos de Concordância Bizantina [BAB 85].

3.1 Classificação de Falhas

Conforme visto no capítulo anterior, um sistema distribuído pode ser definido sob o ponto de vista da rede física ou rede lógica. Sendo esta última mais importante tanto por tratar da aplicação distribuída, como por estar diretamente ligada ao usuário. Para tolerância a falhas a perspectiva do usuário é importante, pois se deseja que a aplicação distribuída continue sua execução mesmo na ocorrência de falhas [JAL 94].

Diz-se que sistema está com defeito (*failure*) quando o serviço fornecido viola a especificação [LAP 85]. Um aspecto interessante a ser verificado é que a definição é

relativa à especificação e não ao projeto. Este detalhe permite que erros inseridos pelo projeto venham a ser considerados no modelo de falha do sistema [BEL 92].

Um erro (*error*) é associado a um defeito do sistema. O sistema está em um estado errôneo quando o processamento de seus algoritmos normais conduz o sistema a um comportamento não especificado. Por algoritmos normais compreende-se o conjunto de algoritmos relacionados com o processamento normal do sistema, excluindo-se os algoritmos associados a outras funções, como por exemplo, de tratamento e recuperação de erros [LAP 85].

Uma falha (*fault*) é a causa direta de um erro. Assim, um erro é a manifestação da falha no sistema e o defeito é a conseqüência do erro no serviço realizado. Esta definição pode ser usada em diferentes níveis de abstração, fazendo com que um defeito em um nível inferior seja considerado como falha em um nível mais alto. Exemplificando, um defeito a nível de processador pode ser considerado como uma falha de sistema operacional [LAP 85].

A percepção de um defeito sugere a presença de um erro, que por sua vez implica a existência de uma falha. Entretanto, a identificação de qual falha provocou um determinado defeito não é trivial, pois falhas distintas podem vir a gerar o mesmo defeito [BEL 92].

Uma maneira possível de classificar falhas em um sistema distribuído é baseada em como os componentes faltosos se comportam quando falham. Tal classificação especifica que suposições podem ser feitas sobre o comportamento do componente no momento da falha [JAL 94].

A semântica de falhas adotada na implementação *do Ambiente para Experimentação e Avaliação de Protocolos de difusão Confiável (ADC)* segue a classificação proposta por [CRI 91]. Esta classificação é baseada em como os componentes faltosos comportam-se no momento da falha. [CRI 91] divide as falhas em quatro categorias: falhas bizantinas, falhas de temporização, falhas por omissão e falhas de *crash*. Estas falhas formam uma hierarquia, representada pela figura 3.1 [BAM 93], onde as falhas de *crash* representam o tipo mais simples e mais restritivo (ou bem definido) e as falhas bizantinas fazem parte do tipo menos restritivo e, porém, mais abrangente.

As *falhas bizantinas*, também conhecidas na literatura como *falhas maliciosas* ou *arbitrárias*, caracterizam-se por apresentar um comportamento totalmente arbitrário, podendo envolver tanto o domínio de valores, como o domínio do tempo. As falhas que envolvem o domínio de valores são conhecidas como *falhas de valor*, enquanto as falhas que envolvem o domínio do tempo são denominadas *falhas de temporização* [CRI 91].

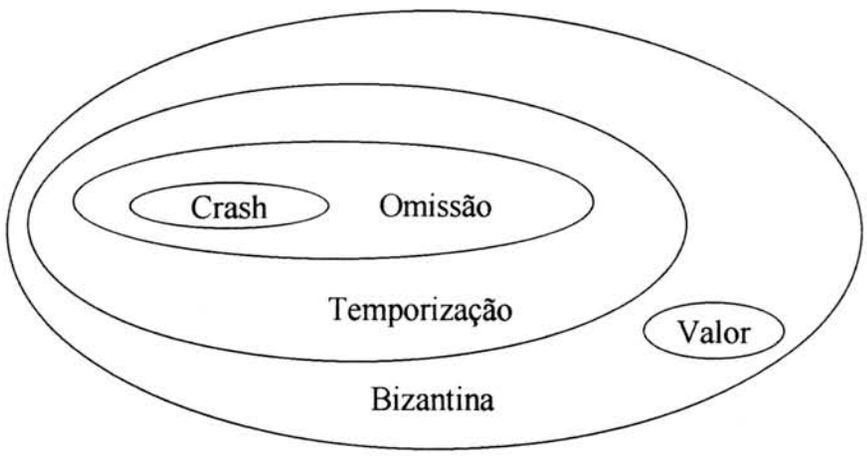


FIGURA 3.1 - Classificação de Falhas [BAR 93c]

As falhas de valor determinam que um serviço entregue no instante esperado apresente seu valor fora da faixa de valores para ele aceitáveis. As falhas de temporização determinam que os valores corretos de um serviço sejam recebidos fora do intervalo de tempo especificado para o mesmo. Sendo assim, as falhas de temporização podem ocorrer por atraso ou por avanço [CRI 91].

As falhas de temporização por atraso, também conhecidas como falhas de *performance*, acontecem quando a recepção do serviço ocorre após um tempo máximo de espera. Já as falhas de temporização por avanço acontecem quando a recepção ocorre antes de um tempo mínimo de espera [CRI 91].

Um caso particular de falhas de temporização é aquele em que uma ativação de serviço nunca tem seu valor liberado, ou seja, quando um servidor omite uma resposta para alguma entrada. Este tipo de falha é conhecido como *falha por omissão*. Se, após a primeira omissão, o serviço deixa de responder sistematicamente a futuras ativações, o serviço é dito incorporando a semântica de *falhas de crash*. Um componente que comete tanto falhas por omissão, como falhas de *crash* não exibe um comportamento incorreto para o exterior [CRI 91].

3.2 Propriedades

Em sistemas onde as mensagens são difundidas por diferentes nodos, há três propriedades que devem ser consideradas [JAL 94]: confiabilidade, ordenação consistente e preservação da causalidade.

Confiabilidade, no contexto de difusão, exige que todas as mensagens disseminadas pela rede sejam recebidas por todos os nodos operacionais. A propriedade de confiabilidade introduz o conceito de atomicidade, onde as ações devem ser

realizadas completamente, ou então não devem ser realizadas de forma alguma [JAL 94].

A ordenação consistente vem complementar a propriedade de confiabilidade exigindo que todas as mensagens enviadas por diferentes nodos sejam entregues numa mesma ordem. Esta ordem, segundo a propriedade de preservação da causalidade deve ser consistente com a relação de causalidade existente entre os eventos [JAL 94]. Destas três propriedades surgem três primitivas de difusão, a saber: difusão confiável, difusão atômica e difusão causal. Existem na literatura diversas classificações para difusão. Neste trabalho, adotou-se a classificação proposta por [JAL 94] tanto por se adaptar às necessidades de implementação do ADC, como por se tratar de uma bibliografia recente.

Difusão confiável suporta apenas a propriedade de confiabilidade, isto é, a mensagem que é difundida é entregue para todos os nodos operacionais, mesmo na ocorrência de falhas no sistema. Difusão atômica, em adição à propriedade de confiabilidade, também suporta a propriedade de ordenação. E difusão causal assegura que a ordem na qual as mensagens são entregues é consistente com a ordenação causal destas mensagens [JAL 94]. Estes três tipos de difusão, bem como a difusão não-ordenada, são examinados em detalhe na próxima seção.

A escolha de uma primitiva é dependente do tipo de aplicação para a qual se destina. Aplicações, cujo envio de mensagens é feito isoladamente, utilizam difusão confiável, a qual suporta somente a propriedade de confiabilidade. Já em aplicações de banco de dados, há necessidade de ordenação das operações realizadas, preservando assim a consistência das informações. Para este tipo de aplicação faz-se uso de difusão atômica. Quando, em uma difusão, os conteúdos das mensagens sendo difundidas dependem de mensagens já enviadas, a relação de causalidade deve ser obedecida, introduzindo a necessidade de difusão causal [JAL 94].

3.3 Tipos de Difusão de Mensagens

As três propriedades vistas anteriormente: confiabilidade, ordenação consistente e preservação da causalidade, bem como os tipos de ordenação de mensagens apresentados na seção 2.1.1: sem ordenação, ordenação FIFO, ordenação causal e ordenação total, conduzem aos seguintes tipos de difusão: difusão não-ordenada, difusão FIFO, difusão atômica, difusão causal e difusão total.

Embora existam, na literatura, diversas classificações para difusão, optou-se pela classificação proposta por [JAL 94] por adaptar-se melhor às exigências do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*.

3.3.1 Difusão não-ordenada

A maneira mais simples de difundir uma mensagem é enviar uma cópia de cada mensagem para todo destinatário individualmente. Esta forma de difusão não oferece qualquer ordenação [JAL 94]. A figura 3.2 ilustra este tipo de difusão.

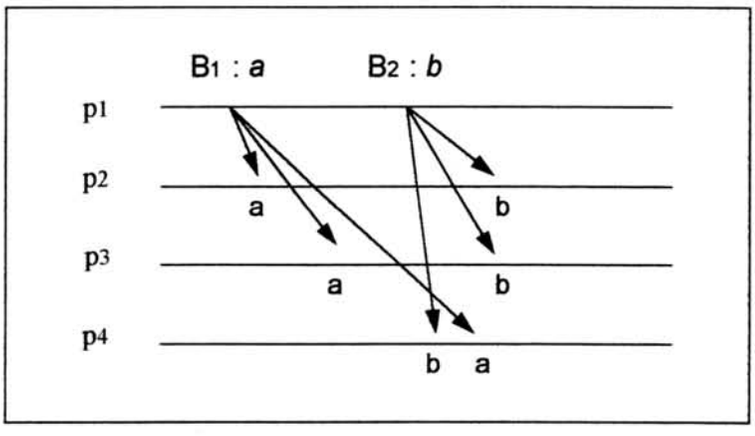


FIGURA 3.2 - Difusão não-ordenada

A figura 3.2 mostra um sistema com quatro processadores: p_1 , p_2 , p_3 e p_4 . O processador p_1 difunde duas mensagens (*a* e *b*) para p_2 , p_3 e p_4 . As duas mensagens chegam na mesma ordem em p_2 e p_3 , mas p_4 as recebe numa ordem diferente.

3.3.2 Difusão FIFO

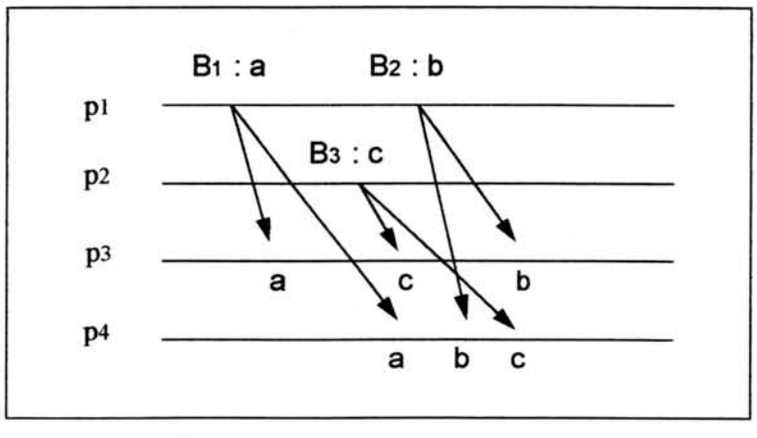


FIGURA 3.3 - Difusão FIFO

Neste tipo de difusão, todas as mensagens difundidas pelo mesmo processador são entregues em qualquer lugar na ordem em que foram enviadas [JAL 94].

A ordenação FIFO, conforme explicado na seção 2.1.1.2, é implementada pela adição de um número de seqüência para toda mensagem [SCH 88] [TAN 89]. Na figura 3.3, o processador p_1 difunde duas mensagens, primeiro a mensagem *a* depois a mensagem *b*. Os processadores p_3 e p_4 recebem as mensagens na ordem em que foram enviadas. As difusões B_2 e B_3 , por serem enviadas por diferentes processadores, podem, então, ser entregues em diferentes ordens em diferentes nodos, como mostra o exemplo.

3.3.3 Difusão Confiável

Por definição, em difusão, um nodo emissor tenta enviar uma mensagem para todos os nodos do sistema. Difusão confiável apresenta uma propriedade básica: toda mensagem de difusão deve ser recebida por todos os nodos operacionais ou, então, não deve ser aceita por nenhum deles. Esta propriedade deve ser preservada mesmo na ocorrência de falhas [AMO 91] [JAL 94].

A difusão confiável encontra aplicações em sistemas onde a ordenação de mensagens não se faz necessária. Tais aplicações compreendem situações onde os processos receptores devem somente apresentar o mesmo conjunto de mensagens ao final da difusão, não importando, portanto, a ordem na qual as mesmas foram recebidas [SIN 94].

Difusão confiável está, geralmente, associada, à difusão FIFO, uma vez que a não exigência de ordenação de mensagens é implementada segundo a ordem de emissão das mesmas (ordem FIFO).

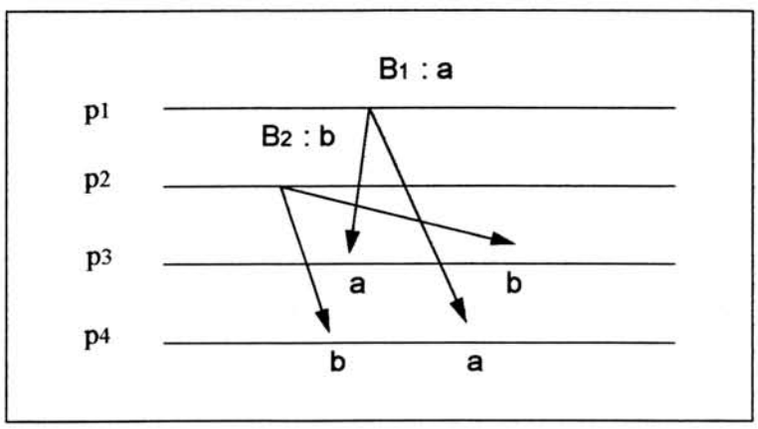


FIGURA 3.4 - Difusão Confiável

Os protocolos que implementam difusão confiável são denominados protocolos de difusão confiável e apresentam um comportamento conforme ilustrado na figura 3.4.

3.3.4 Difusão Atômica

Em comparação aos demais tipos de difusão, a difusão atômica compreende o paradigma de comunicação mais rigoroso. Isto se deve não apenas por exigir que toda mensagem de difusão deve ser recebida por todos os nodos operacionais, ou por nenhum deles, mas pelo fato de exigir também que se múltiplas mensagens são difundidas por diferentes nodos, então estas mensagens devem ser entregues na mesma ordem para todos os nodos. Com isso, em adição à confiabilidade, a propriedade de ordenação consistente também é exigida. Sob o aspecto de tolerância a falhas, ambas as exigências devem ser satisfeitas mesmo na presença de falhas [JAL 94].

Difusão atômica é frequentemente necessária em aplicações como gerenciamento de grupos de processos, dados replicados, processos replicados, etc. Trata-se de uma primitiva bastante útil para construção de sistemas tolerantes a falhas [JAL 94]. Como conseqüência disso, há um grande número de protocolos propostos para atomicamente difundir uma mensagem. Alguns destes protocolos são examinados na seção 3.5. Estes protocolos são chamados de protocolos de difusão atômica. A figura 3.5 ilustra o comportamento destes protocolos.

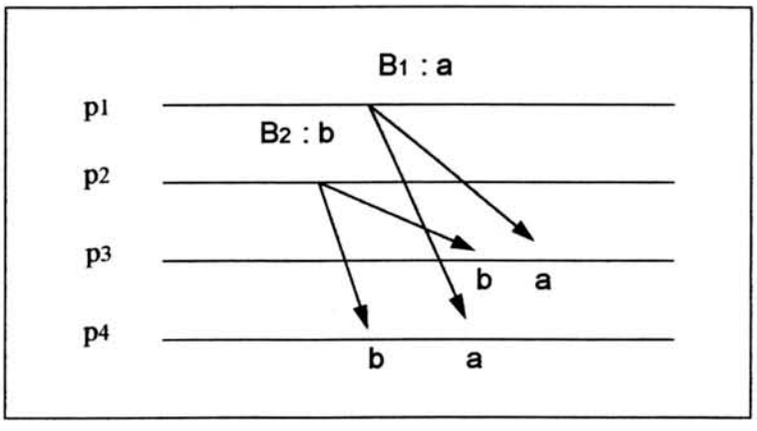


FIGURA 3.5 - Difusão Atômica

Na figura 3.5 são mostradas duas mensagens *a* e *b*, difundidas independentemente por p_1 e p_2 , respectivamente. Ambas mensagens são recebidas na mesma ordem por p_3 e p_4 (primeiro *b*, depois *a*).

3.3.5 Difusão Causal

Devido ao custo inerente do protocolo de difusão atômica é natural procurar protocolos que ofereçam ordenação mais forte que FIFO e que apresentem custo mais baixo que difusão atômica. O protocolo de difusão causal é indicado para esta situação.

Este protocolo é baseado na idéia de causalidade potencial introduzida por Lamport [LAM 78].

O fluxo de informação durante a execução de um sistema distribuído pode ser usado para definir uma ordem parcial na ocorrência dos eventos no sistema [COU 94]. Tais eventos são: o envio de uma mensagem, a recepção de uma mensagem, ou um evento local que afeta apenas um único processador. A figura 3.6 ilustra este fato. Os eventos e_1, e_4, e_{11} e e_{14} são eventos *send*, enquanto e_2, e_5, e_7, e_8, e_{12}, e_{13} e e_{15} são eventos *receive* e e_3, e_9 e e_{10} são eventos locais. De acordo com a definição de Lamport [LAM 78], todos os eventos que são conectados por um caminho neste diagrama são relacionados por causalidade potencial. Tal caminho deve seguir as linhas horizontais (da esquerda para a direita) ou setas de mensagens. Por exemplo, e_{10} é potencialmente causal de e_1, porque há um caminho de e_1 para e_{10} indo através de e_2, e_4 e e_8 (linha pontilhada na figura). Esta dependência é denotada pelo símbolo $\rightarrow$, ou seja, $e_1 \rightarrow e_{10}$.

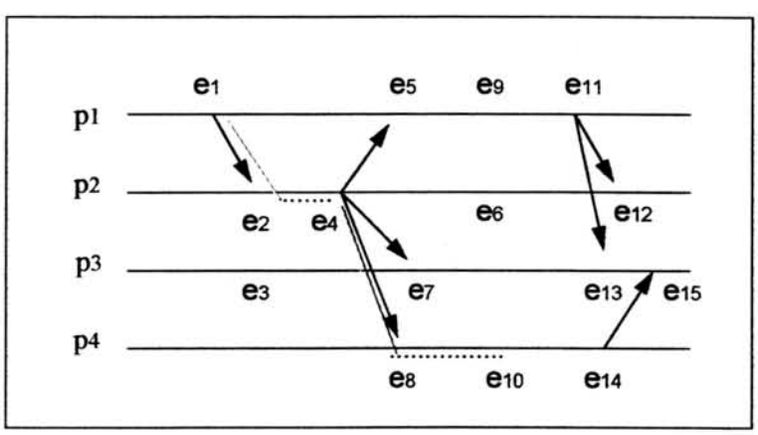


FIGURA 3.6 - Causalidade Potencial

Eventos que não estão conectados por um caminho são chamados eventos concorrentes. Isto é denotado pelo símbolo //. Por exemplo, e_5 // e_{14}. A relação $\rightarrow$ é chamada *causalidade potencial* ou relação de fluxo de informação [LAM 78] [COU 94]. O termo causalidade potencial tem a seguinte explicação: em física, o Princípio da Causalidade diz que uma causa tem que preceder seu efeito. Similarmente, um evento *a* em um processador pode afetar um evento *b* em algum outro processador apenas se há um fluxo de informação de *a* para *b*, isto é, se *a* precede *b* sob $\rightarrow$ [SIL 91].

Portanto pode-se definir a relação de causalidade potencial, também conhecida na literatura como relação "aconteceu-antes" ("*happened-before*"), em um conjunto de eventos, de acordo com as seguintes suposições [SIL 91]:

1. Se A e B são eventos no mesmo processo e o evento A foi executado antes do evento B, então $A \rightarrow B$;
2. Se A é o evento de envio de mensagem por um processo, e B é o evento de recepção desta mensagem por outro processo, então $A \rightarrow B$;
3. Se $A \rightarrow B$ e $B \rightarrow C$, então $A \rightarrow C$.

Enquanto um evento não pode acontecer antes dele mesmo, a relação $\rightarrow$ é uma ordenação parcial irreflexível [LAM 78].

Se dois eventos A e B não estão relacionados pela relação $\rightarrow$, ou seja, A não aconteceu antes de B, e B não aconteceu antes de A, então pode-se afirmar que estes dois eventos foram executados concorrentemente. Neste caso, nenhum evento pode causalmente afetar o outro. Se, contudo, $A \rightarrow B$, então é possível que o evento A afete o evento B causalmente [SIL 91].

Para ser capaz de determinar que um evento A "aconteceu-antes" de um evento B, é necessário que haja um *clock* comum ou um conjunto de *clocks* perfeitamente sincronizados. Como, por definição, em um sistema distribuído nenhum destes está disponível, deve-se definir a relação "aconteceu-antes" sem a utilização de *clocks* físicos. A cada evento do sistema é associado um *timestamp*. Com isso, é possível definir a exigência de ordenação: para cada par de eventos A e B, se $A \rightarrow B$, então o *timestamp* de A deve ser menor que o *timestamp* de B [SIL 91].

Para forçar a exigência de ordenação em um ambiente distribuído deve-se definir dentro de cada processador P_i um *clock* lógico LC_i . O *clock* lógico pode ser implementado como um contador simples, o qual é incrementado entre quaisquer dois eventos sucessivos executados dentro de um processador. Como o *clock* lógico fixa um único número para cada evento no mesmo processador, se um evento A ocorre antes de um evento B no processador P_i, então $LC_i(A) < LC_i(B)$. O *timestamp* para um evento é o valor do *clock* lógico para aquele evento. Logo, este esquema garante que, para quaisquer dois eventos no mesmo processador, a exigência de ordenação é alcançada [SIL 91].

O esquema apresentado anteriormente não garante que a exigência de ordenação seja encontrada entre eventos em processadores distintos. Considerando dois processadores P_1 e P_2 , os quais se comunicam entre si. Supondo que P_1 envia uma mensagem para P_2 (evento A) com $LC_1(A) = 200$, e P_2 recebe a mensagem (evento B) com $LC_2(B) = 195$. Este fato pode ocorrer, por exemplo, porque o processador P_2 é mais lento que o processador P_1 e, portanto, seus *clocks* lógicos apresentam velocidades diferentes. Esta situação viola a exigência de ordenação, uma vez que $A \rightarrow B$, mas o *timestamp* de A é maior que o *timestamp* de B [SIL 91].

Para resolver esta dificuldade, é exigido um procedimento que avance o *clock* lógico de um processador quando ele recebe uma mensagem cujo *timestamp* é maior do

que o valor corrente do seu *clock* lógico. Portanto, se o processador P_i recebe uma mensagem com *timestamp* t e $LC_i(B) < t$, então P_i deve avançar seu *clock*, tal que $LC_i(B) = t + 1$. Assim, no caso do exemplo anterior, quando P_2 recebe a mensagem de P_1, ele avançará seu *clock* lógico tal que $LC_2(B) = 201$. Observa-se ainda que, com o esquema de ordenação por *timestamp*, se os *timestamps* de dois eventos A e B são os mesmos, então os eventos são concorrentes. Neste caso, pode-se usar os números de identidade de cada processador para criar uma ordenação [SIL 91].

As propriedades de ordenação de um protocolo de difusão causal são definidas em termos da informação da relação de fluxo [JAL 94]. Um protocolo de ordenação causal garante que todo processador recebe as mensagens numa ordem que é consistente com $\rightarrow$. Isto é, sempre que dois eventos *send* são relacionados por $\rightarrow$, o protocolo assegura que as duas mensagens são recebidas na mesma ordem em qualquer lugar, dado por $\rightarrow$. Por exemplo, na figura 3.7, difusões B_1 e B_3 são potencialmente causais ($B_1 \rightarrow B_2$, representado pela linha pontilhada). Consequentemente, a mensagem *a* é recebida antes de *c* em ambos os processadores, p_3 e p_4.

Difusões B_3 e B_4, por outro lado, são concorrentes ($B_3 // B_4$). Portanto, as duas mensagens *c* e *d* podem ser recebidas em diferentes ordens em p_3 e p4, como mostrado no exemplo.

Um fato a ser observado é que dois eventos no mesmo processador nunca são concorrentes. Portanto, difusão causal também respeita a ordenação FIFO. Por exemplo, na figura 3.7, $B_1 \rightarrow B_4$, logo a mensagem *a* é recebida em qualquer lugar antes da mensagem *d*.

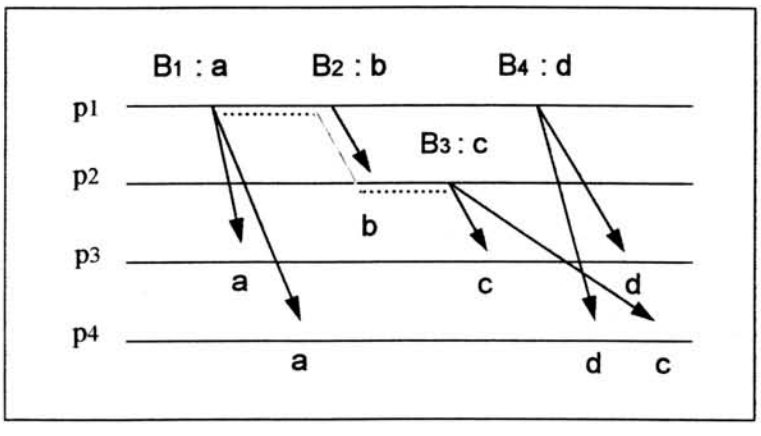


FIGURA 3.7 - Difusão Causal

3.4 Problema de Concordância

Para um sistema distribuído ser considerado confiável é necessário um mecanismo que permita que um conjunto de processos concorde com um valor comum. Há vários motivos pelos quais uma concordância entre todos os nodos do sistema pode não ser alcançada. Primeiro, o meio de comunicação pode ser faltoso, resultando na perda ou corrupção do conteúdo das mensagens. Segundo, os processos podem ser faltosos, resultando em um comportamento não previsível. O melhor que se pode esperar, neste caso, é que processos falhem de forma clara, parando sua execução sem desviar do padrão normal de execução especificado. No pior caso, processos podem enviar mensagens corrompidas para outros processos, ou colaborar com outros processos falhos na tentativa de destruir a integridade do sistema [SIL 91].

Em geral, quando um componente falha, assume-se que ele se comporta de maneira bem definida, apesar de seu comportamento poder ser diferente do seu comportamento livre de falhas. Porém, na maioria dos casos, quando um componente ou um sistema falha, seu comportamento pode ser totalmente arbitrário. Em particular, o sistema ou componente faltoso pode enviar informações totalmente diferentes para os diferentes componentes com os quais ele se comunica. Com tais tipos de falhas, alcançar concordância entre diferentes componentes é difícil [SIL 91].

O problema de alcançar concordância em sistemas distribuídos implica no conhecimento por parte de todos os nodos do sistema sobre os valores recebidos pelos demais nodos [SIN 94]. Este problema é mais conhecido como consenso e encontra larga aplicação em sistemas distribuídos [TUR 92].

Quando o sistema distribuído está livre de falhas, concordância pode ser facilmente alcançada entre seus nodos. Nesta situação, cada nodo divulga seus valores para os demais nodos e, cada um deles deve tomar uma decisão sobre os valores recebidos através de votação pela maioria. Contudo, quando o sistema está propenso a falhas, este método não funciona. Isto ocorre porque os nodos faltosos do sistema podem enviar valores conflitantes para os demais, evitando, com isso, a obtenção da concordância entre os nodos não faltosos sobre os valores divulgados. Na presença de falhas, os nodos de um sistema distribuído devem trocar seus valores entre si, retransmitindo os valores recebidos pelos demais. Com isso consegue-se isolar os efeitos dos nodos faltosos sobre os valores corretamente difundidos no sistema [SIN 94].

Logo, em problemas de concordância, os nodos não-faltosos de um sistema distribuído devem ser capazes de chegar a um comum acordo, mesmo se determinados componentes do sistema sejam faltosos. O comum acordo é alcançado através de um protocolo de concordância, o qual envolve várias rodadas de trocas de mensagem entre todos os nodos do sistema [SIN 94].

3.4.1 O Problema dos Generais Bizantinos

[CRI 91] utiliza o termo *semântica de falhas bizantinas* para descrever um serviço que exibe todas as semânticas de falhas, a saber: falhas de *crash*, falhas por omissão, falhas de temporização e falhas de valor. O termo comportamento de falhas bizantino é comumente utilizado para descrever a pior semântica de falhas possível de um nodo. Por pior semântica de falhas entende-se a menos restritiva e, portanto a mais abrangente de acordo com a classificação de [CRI 91] apresentada em 3.1.

[LAM 82] formulou o Problema dos Generais Bizantinos, o qual modela uma situação em que muitos nodos trabalham corretamente, porém alguns nodos apresentam um comportamento arbitrário, os quais incorporam a semântica de falhas bizantinas, ou maliciosas, ou ainda arbitrárias. Nodos sujeitos a falhas bizantinas podem, por exemplo, enviar mensagens contraditórias para receptores ou impedir que uma mensagem correta chegue a seu destino.

O problema dos Generais Bizantinos descreve uma situação na qual n divisões de um exército Bizantino estão acampadas ao redor de uma cidade inimiga. Cada divisão do exército é comandada por um general. As várias divisões estão geograficamente dispersas e os generais podem se comunicar entre si somente por troca de mensagens. Cada general tem que votar sim ou não para um plano de ação. É de extrema importância que todos os generais concordem com a mesma ação, uma vez que um ataque por somente algumas das divisões resultaria em derrota. Contudo, alguns dos generais podem ser traidores, evitando então que os generais leais cheguem a um acordo sobre que atitude deve ser tomada. Um algoritmo resolve o problema dos Generais Bizantinos se ele consegue fazer com que todos os generais leais concordem com um valor de decisão dentro de um intervalo de tempo limitado [COU 94].

O principal objetivo do Problema dos Generais Bizantinos é a obtenção de uma concordância entre todos os nodos não-faltosos. Esta concordância entre todos os nodos do sistema distribuído livres de falha é também denominada consenso [TUR 92] [BAB 87] [FIS 86].

Para a determinação do consenso, cada nodo deve tomar uma decisão sobre um valor de difusão, com base nos valores recebidos dos outros nodos do sistema. Para tanto, exige-se que todos os nodos não-faltosos tomem a mesma decisão. Com isso, o objetivo é assegurar que todos os nodos não-faltosos apresentem o mesmo conjunto de valores. Se todos os nodos não-faltosos recebem o mesmo conjunto de valores dos diferentes nodos do sistema, consenso pode ser facilmente alcançado [JAL 94].

Conforme mostrado em [FIS 85], se o sistema distribuído é assíncrono, ou seja, se não há limite na velocidade relativa dos nodos ou atrasos de comunicação, o consenso é impossível mesmo que haja apenas um nodo falho e, mesmo que o tipo de falha deste nodo seja *crash*. Isto porque, em sistemas assíncronos, a falha de um nodos para enviar uma mensagem não pode ser distinguida da situação onde o nodo e a rede

de comunicação são extremamente lentos. Em tal situação, um nodo nunca pode detectar a ausência de uma mensagem com certeza. Consequentemente, um nodo faltoso pode bloquear o algoritmo de consenso simplesmente por não enviar uma mensagem. Portanto, muitos dos algoritmos propostos para resolver o problema dos Generais Bizantinos para sistemas distribuídos são síncronos. Em sistemas síncronos, tanto os atrasos de mensagens como as diferenças nas velocidades relativas dos processos são limitados [JAL 94].

O problema de concordância é difícil porque a informação enviada por um nodo não pode ser confiável, uma vez que o sistema adota a semântica de falhas menos restritiva, ou seja, falhas bizantinas. Desta forma, para um nodo n_1 concordar com um valor enviado por outro nodo n_2 , além de armazenar o valor recebido de n_2 , o nodo n_1 deve comparar este valor com os valores recebidos dos demais nodos do sistema. Com isto, é possível fazer uma verificação sobre o valor original. O problema torna-se complexo, uma vez que nesta difusão podem haver nodos faltosos, os quais se comportam de forma arbitrária. Uma solução para este problema seria a limitação no número de nodos faltosos do sistema [JAL 94].

Considerando-se um sistema com três nodos, sendo um destes faltoso, o problema não pode ser resolvido através de trocas de mensagens simples [LAM 82]. Supondo que os nodos necessitem concordar com um valor de decisão: S para sim, e N para não, considera-se as figuras 3.8 e 3.9 [JAL 94].

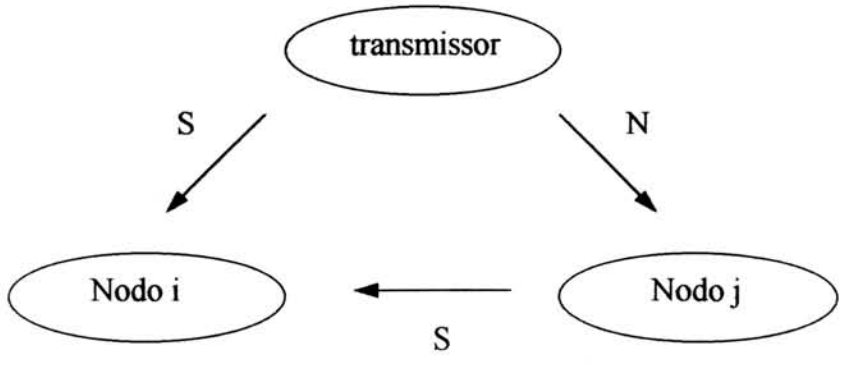


FIGURA 3.8 - Nodo j é faltoso [JAL 94]

Na figura 3.8, um dos nodos receptores (j) é faltoso e, embora o emissor envie para ele o valor N, ele transmite para o nodo i que recebeu o valor S do emissor. Na figura 3.9, o nodo emissor é faltoso e envia N para i e S para j, o qual difunde seu valor para i. Ambas as situações são indistingüíveis para i, e i não pode decidir se o valor enviado pelo emissor pode ser considerado correto ou se o valor de j é correto.

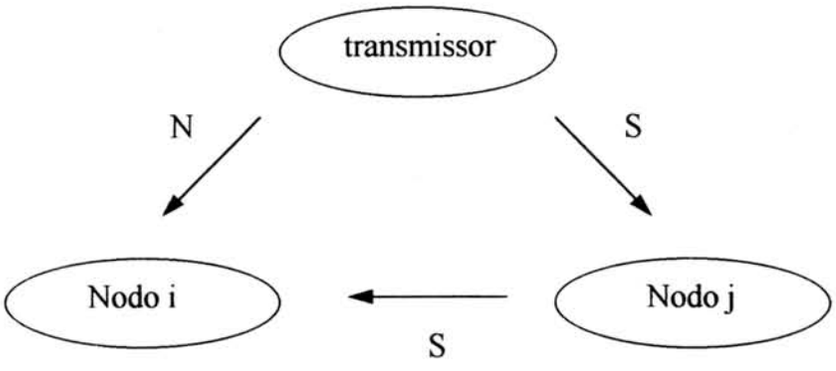


FIGURA 3.9 - Transmissor é faltoso [JAL 94]

De acordo com o exemplo explicado anteriormente [LAM 82] e [PEA 80] mostraram que com mensagens simples é impossível resolver este problema a menos que 2/3 dos nodos sejam não-faltosos. Isto é, o número de nodos faltosos tem que ser menor do que 1/3 do total do número de nodos do sistema. Para competir com m nodos faltosos, pelo menos $3m+1$ nodos são necessários para alcançar o consenso.

O problema é simplificado se as mensagens podem ser assinadas. Isto é, um nodo pode adicionar um assinatura para cada mensagem que ele envia. O conteúdo da mensagem com assinatura não pode ser alterado por um nodo faltoso sem que seja detectada a alteração pelos nodos não-faltosos [JAL 94].

[LAM 82] propôs dois protocolos para resolver o problema dos Generais Bizantinos, um protocolo com mensagens simples e um protocolo com mensagens assinadas.

O primeiro protocolo utiliza mensagens simples e respeita a condição de que consenso pode ser alcançado na presença de m nodos faltosos se e somente se o total de nodos é menor ou igual a $3m + 1$.

O protocolo trabalha em rodadas, cada rodada consistindo de trocas de mensagens entre os nodos. Na primeira rodada, o transmissor envia valores para os demais nodos do sistema. Os nodos receptores não podem confiar nos valores recebidos pelo transmissor, uma vez que o mesmo pode ser faltoso. Então, cada receptor coleta informações sobre os valores recebidos pelos demais. De acordo com as informações recebidas, cada nodo realiza uma votação e decide pelo valor da maioria como sendo o valor recebido originalmente pelo transmissor.

Na segunda rodada, cada receptor age como transmissor e envia mensagens para os outros nodos, exceto para o transmissor. A transmissão das mensagens da segunda rodada tem como objetivo informar a todos os nodos o valor que cada um recebeu do transmissor. Entretanto, os receptores não podem confiar nos valores recebidos na

segunda rodada, pelo mesmo motivo que não podiam na rodada anterior. Portanto, a desconfiança na rodada 2 exige a rodada 3. Na rodada 3, os mesmos procedimentos são executados e, novamente, não se pode confiar nos valores recebidos. Lamport provou que são necessárias $m + 1$ rodadas de protocolo para chegar a um consenso, onde m é o número de nodos faltosos.

O segundo protocolo trabalha com mensagens assinadas. Para cada mensagem enviada, deve ser adicionada a assinatura do emissor. Com isso, qualquer alteração no conteúdo da mensagem pode ser detectada. Ao contrário do protocolo anterior, o protocolo com mensagens assinadas consegue alcançar consenso com um número arbitrário de nodos faltosos.

No protocolo, um nodo transmissor envia uma mensagem assinada para os outros nodos. Cada nodo adiciona assinatura à mensagem recebida e a retransmite para todos na próxima rodada de troca de mensagens. Se o receptor é não-faltoso, sua mensagem terá o mesmo conteúdo que a mensagem que ele recebeu do transmissor. Se o receptor é faltoso, ele deve também enviar a mensagem contendo o mesmo valor, ou não envia nada. O caso em que o receptor faltoso altera a mensagem e é descoberto pelos receptores pode ser modelado como o não envio da mensagem. O número de rodadas necessárias para a determinação do consenso também é $m + 1$.

De acordo com a discussão de [JAL 94], ambos os protocolos trabalham em rodadas e apresentam um custo bastante elevado em termos de número de rodadas e número de mensagens necessárias para a determinação do consenso.

O primeiro protocolo exige $m + 1$ rodadas de troca de mensagens. Foi mostrado que nenhum algoritmo determinístico para concordância bizantina pode alcançar consenso em menos do que $m + 1$ rodadas. Portanto, em termos do número de rodadas o primeiro protocolo é razoável. Contudo, o número de mensagens exigido é exponencial. Da mesma forma, o segundo protocolo exige $m + 1$ rodadas, o que também é razoável, e igualmente ao anterior exige um número exponencial de mensagens.

As soluções propostas anteriormente são determinísticas e garantem consenso com um determinado custo em termos de número de rodadas e número de mensagens trocadas. A impossibilidade de ter protocolo de concordância determinístico em sistemas distribuídos assíncronos conduz a soluções randômicas [JAL 94].

Os protocolos randômicos utilizam-se de números aleatórios, os quais são usados por todos os nodos do sistema para tentar alcançar concordância. Assim como em protocolos determinísticos, os protocolos randômicos também trabalham em rodadas. Contudo, não há limite exato no número de rodadas necessárias para alcançar concordância. Geralmente, protocolos randômicos trabalham com um número esperado de rodadas necessárias para a determinação do consenso. A vantagem destes protocolos é o número esperado de rodadas necessárias para que se chegue a um consenso é

consideravelmente menor do que os números apresentados pelos protocolos determinísticos [JAL 94].

Os generais bizantinos são ótimos em relação ao número de rodadas para a semântica de falhas para a qual se propõe (falhas bizantinas). Em sistemas reais, pode-se adotar uma semântica de falhas mais restritiva, conseguindo-se assim minimizar o número de rodadas e diminuir consideravelmente o número de mensagens. Obviamente, uma semântica de falhas mais restritiva deve ser adotada com cuidados. Ou os nodos do sistema são construídos para garantir a semântica, ou o sistema vai operar com baixa confiabilidade. Neste contexto, protocolos de difusão confiável surgem como alternativa para solucionar o problema de consenso em sistemas distribuídos, de modo que o número de mensagens trocadas seja minimizado, não comprometendo, no entanto, a confiabilidade do sistema.

O problema de concordância em sistemas distribuídos tem sido estudado sob diversos aspectos. [SIN 94] classificou os problemas de concordância em três categorias, a saber: Problema de Concordância Bizantina, Problema do Consenso e Problema de Consistência Interativa.

3.5 Protocolos de Difusão Confiável

Um protocolo de difusão é um serviço responsável por garantir, mesmo na presença de falhas, um comportamento bem definido em relação a mensagens difundidas no sistema [SEG 83].

Em um sistema distribuído pode-se ter vários grupos, sobrepostos ou não, segundo as funcionalidades definidas no sistema, suportadas pelo serviço de comunicação do grupo. As mensagens recebidas em uma unidade do grupo somente são consideradas aceitas, e passadas ao usuário do serviço, quando satisfazem todas as condições do protocolo de difusão. Esse ato de aceitação é chamado de compromisso da mensagem (*"message commitment"*) e dependerá, portanto, das propriedades envolvidas no protocolo utilizado [DUE 92].

Atualmente, existe na literatura um vasto número de proposições de protocolos de difusão. A escolha do protocolo depende da especificação das falhas a serem suportadas e o tratamento destas. Quanto mais restritivas forem as hipóteses de falhas, menos complexo e de menor custo é a concepção e a implementação dos algoritmos dos serviços considerados.

No estudo realizado em [BAR 94] como preparação a essa dissertação, foram classificados os protocolos de difusão propostos por [BAB 86], [BIR 87], [CHA 84] e [CRI 85], os quais obedecem as propriedades descritas na seção 3.2. A seguir, conforme

[BAR 94], é apresentada uma síntese das principais características de protocolos mais representativos.

3.5.1 Protocolo de Birman

Birman [BIR 87] propôs dois protocolos pertencentes à classe de protocolo com comunicação de grupo assíncrona, os quais suportam falhas de *crash*. A distinção entre os dois protocolos reside no tipo de ordenação de mensagens utilizado.

O primeiro protocolo utiliza ordenação total de mensagens e é implementado em três fases. Todo processador mantém uma fila de mensagens entregues. Quando uma mensagem de difusão é recebida (primeira fase), ela é adicionada à fila, mas ainda não é entregue para o programa de aplicação. O receptor fixa um número de prioridade temporário para a mensagem e retorna este número para o emissor da difusão (segunda fase). O receptor escolhe este número sendo o maior de todos os números fixados para as mensagens correntes na fila ou previamente entregues. O emissor coleciona todos os números de prioridade, computa seu máximo e envia este número para todos os nodos destinatários (terceira fase). Cada receptor troca o número de prioridade temporário pelo número recém recebido e reordena a fila. Uma mensagem é entregue quando ela tiver recebido seu número de prioridade final e nenhuma mensagem com número de prioridade menor está na fila.

O protocolo acima descrito se adapta às exigências de confiabilidade e ordenação consistente, sendo então classificado como protocolo de difusão atômica [JAL 94]. O protocolo apresenta ainda funções para informar aos membros operacionais do grupo de processos quando um membro falhou, reintegrou-se, aderiu ou saiu voluntariamente, ou ainda, quando houve alguma mudança nas propriedades globais do sistema distribuído.

O segundo protocolo porposto por Birman considera a preservação da propriedade de causalidade potencial na ordenação das mensagens. Birman implementa ordenação causal restringindo a necessidade de ordenação às mensagens que apresentam alguma relação de causa e efeito. De acordo com [JAL 94] este protocolo é classificado como protocolo de difusão causal. Para cada mensagem de difusão enviada de um nodo emissor para um nodo receptor, o nodo emissor envia todas as mensagens contidas em seu *buffer*, as quais casualmente precedem a mensagem de difusão enviada.

3.5.2 Protocolo de Chang

O protocolo de Chang [CHA 84] apresenta características de protocolo assíncrono com ordenação total, suportando falhas por omissão e obedecendo a

exigência de confiabilidade. De acordo com estas caraterísticas, o protocolo pode ser classificado como protocolo de difusão atômica [JAL 94].

O *commit* de mensagem, ou seja, a aceitação da mensagem, é centralizado sobre o conceito de *token* circulante. A estação possuidora do *token* é denominada *token site*, e é responsável, enquanto possuidora do *token*, pelo reconhecimento das mensagens difundidas, pela retransmissão de mensagens, quando requisitado por algum receptor e também, pela transferência do *token* para o próximo *token site*. As estações operacionais do sistema são relacionadas em uma *token list* que determina o anel virtual para a transferência do *token* [AMO 91].

No protocolo de Chang toda mensagem é difundida em duas fases. Um processador desejando difundir uma mensagem envia esta para um processador distinto, o *token* (primeira fase). O *token* então difunde a mensagem para seus destinatários (segunda fase). Desta forma todas as mensagens de difusão são entregues na ordem que forem recebidas e difundidas pelo emissor.

3.5.3 Protocolo de Cristian

Os protocolos propostos por [CRI 85] representam uma família de protocolos de difusão atômica que resistem a erros de severidade progressiva. Estes protocolos baseiam-se em difusão sobre uma rede ponto-a-ponto de redundância espacial. As propriedades de atomicidade, ordenação total e terminação são suportadas através de uma execução fortemente síncrona dada pela sincronização dos relógios locais.

Os protocolos síncronos (e/ou fortemente síncronos), ao invés de trocas de mensagens-protocolo, utilizam a passagem de tempo e o conhecimento dos limites temporais para obter as informações para o acordo e a ordenação, baseados no comportamento de pior caso. Assim, todas as mensagens são engajadas simultaneamente na mesma ordem por todos os receptores, com base no tempo t em que as mensagens foram enviadas e no atraso máximo que uma mensagem pode levar para alcançar todos os seus destinos [DUE 92].

3.5.4 Protocolo de Babaoglu

Os protocolos propostos por Babaoglu [BAB 85] fazem parte da classe de protocolos síncronos, com ordenação de mensagens do tipo FIFO, suportando a propriedade de confiabilidade na entrega das mensagens. Estes protocolos suportam falhas de *crash*, falhas por omissão e falhas bizantinas. Os protocolos de [BAB 85] utilizam múltiplos canais de comunicação independentes, por meio dos quais as mensagens são difundidas.

Os protocolos de [BAB 85], de acordo com a característica intrínseca dos sistemas distribuídos síncronos, utilizam a limitação no tempo de entrega das mensagens para implementar difusão confiável em apenas duas rodadas de execução.

3.5.5 Considerações Gerais

Ordenação atômica torna o projeto de aplicações distribuídas tolerantes a falhas mais fácil porque reduz a incerteza causada por atrasos de mensagens e falhas no sistema. Contudo, este benefício não vem sem o custo. Os protocolos descritos em 3.5.1 e 3.5.2 necessitam de duas ou três fases de comunicação antes de qualquer mensagem de difusão atômica poder ser entregue [JAL 94].

Os protocolos de difusão propostos por Babaoglu são capazes de suportar falhas bizantinas e falhas por omissão, as quais incluem falhas de *crash*. Babaoglu descreve protocolos que não satisfazem a exigência de ordenação consistente, uma vez que a aceitação de mensagens é feita obedecendo a ordem FIFO. Os protocolos seguem a primitiva de difusão confiável, já que suportam somente a propriedade de confiabilidade.

Os protocolos de Birman, Chang e Cristian obedecem as exigências de confiabilidade e ordenação consistente. [BIR 87] propôs dois tipos de protocolo, o primeiro satisfaz à exigência de confiabilidade, sendo classificado como protocolo de difusão atômica, o segundo preserva a relação de causalidade no envio das mensagens, seguindo a primitiva de difusão causal. Uma das características comuns entre os dois protocolos de [BIR 87] é o fato de ambos suportarem a semântica de falhas de *crash*.

TABELA 3.1- Características dos Protocolos de Difusão Confiável [BAR 94]

	BIRMAN	**CHANG**	**CRISTIAN**	**BABAOGLU**
Terminação	assíncrona	assíncrona	síncrona	síncrona
Ordenação	total ou causal	total	total	FIFO
Acordo	distribuído	centralizado ou distribuído	distribuído	distribuído
Falhas suportadas	crash	omissão	omissão ,..., bizantinas	omissão e bizantinas
Desempenho	2 rodadas de troca de msg	L transferências de token	Δ fixo	2 rodadas de execução
Particionamento da rede	não admite	não admite	não admite	admite

O protocolo de Chang é capaz de suportar tanto falhas por omissão como falhas de *crash*, enquanto os protocolos propostos por Cristian abrangem desde falhas de *crash* até falhas bizantinas.

As características dos protocolos de difusão confiável sumariamente descritos nas seções 3.5.1 à 3.5.4 encontram-se sintetizadas na tabela 3.1 [BAR 94].

4 Plataforma de Desenvolvimento do ADC

O ADC (*Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável*) foi desenvolvido em estações de trabalho SUN (*Unix*), utilizando o sistema *HetNOS*.

O *HetNOS* (*Het*erogeneous *N*etwork *O*perating *S*ystem) foi desenvolvido como dissertação de mestrado por [BAA 93] no CPGCC da UFRGS no período de 1991 à 1993. Ainda em 1990 e 1991 foi iniciado o projeto do HetNOS como *software* distribuído com funções de comunicação entre processos remotos. Atualmente, o HetNOS encontra-se entre as fases de depuração e otimizações.

O HetNOS é um sistema operacional de rede heterogêneo orientado à programação de aplicações paralelas e distribuídas. Através de um núcleo distribuído executado em cada máquina, o sistema HetNOS fornece um ambiente de alto nível para programação distribuída em uma rede de estações de trabalho [BAA 93].

Aplicações HetNOS podem ser programadas para operar de forma distribuída visando aumentar seu desempenho e/ou confiabilidade. O ambiente de programação propiciado pelo HetNOS pode ser considerado uma extensão daquele fornecido pelo sistema operacional nativo, o *Unix* [BAA 93].

Aplicações distribuídas podem ser divididas em módulos independentes que executam de forma parcialmente paralela em relação aos demais. Um módulo, que é um fluxo independente de execução, pode ser denominado *task* ou tarefa. Em sistemas distribuídos, tais módulos são implementados como processos independentes, de forma que as partes de uma aplicação possam executar em diferentes máquinas. Como cada módulo é implementado como um processo independente, há necessidade de um paradigma de comunicação e interação entre os mesmos. O paradigma de comunicação utilizado pelo HetNOS é a troca de mensagens. O HetNOS fornece um conjunto de funções específicas para troca de mensagens [BAA 95].

A comunicação e sincronização entre os processos no HetNOS é obtida através do acesso a um conjunto de primitivas de alto nível, implementadas através de funções contidas em um conjunto de bibliotecas. Estas, por sua vez, devem estar ligadas às aplicações, e são utilizadas para acessar de forma rápida e fácil os serviços prestados pelos módulos do HetNOS [BAA 95].

Esse capítulo apresenta os recursos do HetNOS usados para o desenvolvimento do ADC. Para o leitor não interessado nas características do HetNOS sugere-se apenas a leitura do item 4.5.

4.1 Paradigma de Comunicação do HetNOS

O paradigma de comunicação entre processos é decisivo no projeto da aplicação distribuída. Dentre os paradigmas conhecidos para comunicação entre processos, destacam-se em ordem crescente de abstração: troca de mensagens, chamada remota de procedimento (RPC) e memória compartilhada distribuída (DSM). Entretanto, como regra geral, o grau de abstração e sofisticação do ambiente é inversamente proporcional ao seu desempenho [BAL 89].

No paradigma de troca de mensagens, a primitiva de comunicação mais elementar é o envio de uma mensagem de um processo origem (*sender*) para um processo receptor (*receiver*). A implementação do sistema operacional (ou das linguagens) usualmente oferece um serviço confiável de entrega de mensagens, que se encarrega de repetir mensagens erradas ou não entregues [BAL 89].

Com RPC a comunicação é implementada através de chamadas a procedimentos remotos, sendo os detalhes de comunicação, como transmissão de pacotes, escondidos em rotinas denominadas *stubs*, as quais são geradas automaticamente [BAL 89].

No paradigma de memória compartilhada distribuída, há um sistema que faz com que o programador não tome conhecimento da distribuição da memória física. Este sistema pode ser tanto o *run-time* da linguagem, como um servidor. Uma possibilidade é implementar a memória como um espaço de *tuplas*, fornecendo primitivas de acesso a essas tuplas [BAL 89].

Cada uma das opções apresenta suas vantagens e desvantagens em relação às demais, não existindo consenso sobre qual seja o melhor paradigma [BAL 89]. Na realidade, é possível que não exista um "melhor paradigma", acredita-se que a questão é relativa à classe de problema ao qual o paradigma será aplicado, ou seja, é dependente do tipo de aplicação [BAA 95a].

O HetNOS adota o paradigma de troca de mensagens. A esolha é justificada pela flexibilidade e desempenho oferecidos por essa abordagem. Com o paradigma de troca de mensagens, é possível implementar os paradigmas de comunicação RPC e memória compartilhada distribuída. Dos três paradigmas de comunicação citados, o de troca de mensagens correspode ao de mais baixo nível. Essa desvantagem é amenizada pela simplicidade da interface do HetNOS. Uma gama de primitivas é colocada à disposição do usuário, permitindo diversos tipos de comunicação e sincronização [BAA 95a].

O HetNOS é mais flexível que o RPC, ao mesmo tempo que exime o usuário da gerência associada à troca de mensagens com *sockets* ou TLI. Processos enviam e recebem mensagens através de sua identificação, que é única no sistema como um todo. Ao identificar o destino de uma mensagem, um processo designa apenas o nome de um ou mais processos (independente de localização). Não há estabelecimento ou

encerramento de conexões, nem a necessidade de determinar protocolos ou gerenciar portas de comunicação [BAA 95a].

4.2 Troca de Mensagens

Conforme já foi mencionado, o paradigma de comunicação adotado pelo HetNOS é a troca de mensagens. Para simplificar as primitivas de comunicação, as mensagens não seguem um tipo estruturado de dado. Mensagens, no HetNOS, são cadeias de caracteres (*strings* em C), ou seja, pacotes de tamanho variado, contendo uma seqüência de *bytes*, terminados por um caractere nulo. Em geral, o uso de *strings* como mensagens resulta em simplicidade e flexibilidade, o que compensa a queda de desempenho introduzida pelo método [BAA 93].

A comunicação por mensagens ocorre através de variáveis do par básico *send/receive*. Na emissão de uma mensagem, um processo origem envia (com *send*) uma mensagem a um processo destino. Esta mensagem é armazenada em um *buffer* do HetNOS até que o destino esteja pronto para recebê-la. Esta prontidão é indicada por uma das primitivas de *receive*. Ao enviar uma mensagem, são fornecidos dois argumentos: o nome do processo destino e o conteúdo da mensagem a ser enviada. Para receber uma mensagem, deve-se indicar o nome de um processo origem e uma área onde a mensagem deve ser armazenada [BAA 95a].

É possível que um processo (tal como um servidor) queira aceitar mensagens de qualquer processo ("ANY"). Neste caso, o receptor pode identificar seguramente o nome do processo origem, pois a função retorna um apontador para uma área interna à função, contendo o nome do origem. Este nome fica disponível até o próximo *receive* desse mesmo processo [BAA 95a].

4.2.1 Classes de Primitivas Disponíveis

TABELA 4.1 - Características das Primitivas de Envio

bloqueantes	o emissor somente é liberado, no mínimo, após a chegada de um acknowledgement (ACK) do nodo receptor
assíncronas	o emissor é liberado imediatamente, ou então, assim que chega o ACK
síncronas	o emissor é liberado apenas após chegar o ACK do nodo destino indicando que a mensagem foi lida pelo processo destino
bufferizadas	caso o destino não esteja pronto para receber a mensagem, a mesma é inserida em uma fila associada ao receptor, a qual é mantida pelo sistema
não-bufferizadas	caso o processo receptor não esteja pronto para ler a mensagem, a mesma não é bufferizada e o processo origem é informado do erro

Um problema sério é a taxonomia quanto às primitivas de comunicação. As características das primitivas de envio estão descritas na tabela 4.1 [BAA 95a].

Atualmente todas as primitivas de envio do HetNOS são bloqueantes, pois são confiáveis. Sempre um ACK do nodo destino é esperado antes de se liberar o processo requisiante do *send*. Há primitivas assíncronas, como *h_send( )*, onde o processo origem é liberado assim que uma confirmação de entrega da mensagem no nodo destino é recebida, e síncronas, como *h_sync_send( )*, onde o processo origem é liberado apenas quando uma confirmação de que o processo leu a mensagem é recebida [BAA 95a].

Mensagens são sempre encaminhadas ao nodo destino (*bufferização* no nodo destino). Quando o processo destino não está esperando por esta mensagem, a mesma é inserida em uma fila associada ao destino. Mais tarde a mensagem poderá ser lida pelo destino. Estas primitivas de envio são denominadas *bufferizadas*. O HetNOS possui uma primitiva de envio *não-bufferizada*: quando o destino não está pronto para receber a mensagem, ela é descartada e o origem é avisado que a mensagem não pôde ser entregue porque o destino não estava esperando. A Tabela 4.2 ilustra as primitivas de envio de mensagens [BAA 95a].

TABELA 4.2 - Primitivas de Envio de Mensagens [BAA 95a]

`int h_send(char *dest, char *msg)` envia de forma **assíncrona** a cadeia em `msg` ao processo `dest`
`int h_sync_send(char *dest, char *msg)` envia de forma **síncrona** a cadeia em `msg` ao processo `dest`
`int h_nonblk_send(char *dest, char *msg)` envia a cadeia em `msg` como mensagem ao processo `dest`

TABELA 4.3 - Características das Primitivas de Recepção

bloqueantes	o receptor espera o término da operação, não podendo realizar qualquer processamento após o pedido de recepção
não-bloqueantes	o receptor requisita a recepção e retorna imediatamente. Caso uma mensagem não esteja disponível para leitura, o pedido pode ficar pendente ou uma condição de erro é retornada ao usuário
síncronas	o receptor fica bloqueado, sendo liberado apenas após uma mensagem "aceitável" ter sido lida
assíncronas com interrupção	requer a recepção de uma mensagem e retorna ao processamento normal. Quando uma mensagem chega, o processo receptor é interrompido e é invocado um handler (tratamento da mensagem)
assíncronas com pooling	como acima, mas o receptor deve continuamente verificar se há mensagem disponível

A tabela 4.3 apresenta as principais características das primitivas de recepção [BAA 95a].

As primitivas de recepção do HetNOS são essencialmente bloqueantes, retornando apenas quando uma mensagem estiver disponível, como ocorre com *h_recv( )*. O HetNOS dispõe também de uma primitiva de recepção não-bloqueante, denominada *h_nonblk_recv( )*. Nesta primitiva, o processo receptor recebe a mensagem apenas se ela já estiver disponível. Caso contrário, é retornado um código de erro indicando que o processo será bloqueado. Não há uma primitiva assíncrona, devido à complexidade de sua implementação e de seu uso. Todas as funções de recepção fornecem como valor de retorno o nome do processo origem e armazenam a mensagem recebida em um *buffer* que deve ser criado pelo usuário. A Tabela 4.4 ilustra as primitivas de recepção de mesagens disponíveis [BAA 95a].

TABELA 4.4 - Primitivas de Recepção de Mensagens [BAA 95a]

`char *h_recv(char *orig, const char *msg)` solicita e espera o recebimento de uma mensagem em `msg`, de acordo com o origem especificado em `orig`
`char *h_nonblk_recv(char *orig, const char *msg)` solicita e espera o recebimento de uma mensagem em `msg`, de acordo com o origem especificado em `orig`, mas não bloqueia se não houver uma mensagem disponível

4.2.2 Primitivas de Comunicação Multiponto

O HetNOS oferece um conjunto de primitivas *multicast* para os casos em que é necessária a comunicação entre um conjunto de processos. Tais primitivas permitem enviar, de uma vez só, uma única mensagem a uma lista de processos. Estas mensagens podem ser recebidas com primitivas não multiponto, como *h_recv( )* [BAA 95a].

TABELA 4.5 - Primitivas de Comunicação Multiponto [BAA 95a]

`int h_multi_send(char *dest_list, char *msg)` envia de forma **assíncrona** a cadeia em `msg` como mensagem à lista de processos em `dest_list` (nomes de processos separados por espaços)
`int h_multi_sync_send(char *dest_list, char *msg)` envia de forma **síncrona** a cadeia em `msg` como mensagem à lista de processos em `dest_list` (nomes de processos separados por espaços)
`char *h_multi_recv(char *lista_origens, const char *msg)` solicita e espera o recebimento de uma mensagem em `msg` que pode ter sido enviada por qualquer um dos processos mencionados na lista de processos

Para receber uma mensagem de um conjunto de processos, retornando após a primeira mensagem destes processos que estiver disponível, utiliza-se *h_multi_recv( )*. Assim como as demais funções de recepção, a primitiva retorna um apontador para uma área interna contendo o nome do processo que enviou a mensagem efetivamente recebida. As primitivas de comunicação multiponto estão descritas na tabela 4.5 [BAA 95a].

As funções de troca de mensagens multiponto podem ser acessadas indiretamente através das funções tradicionais ponto-a-ponto. Basta fornecer à uma mensagem ponto-a-ponto uma lista de processos como origem ou destino [BAA 95a].

4.3 Gerência de Processos

Um dos aspectos mais importantes em termos de programação distribuída é a gerência de processos. Com *sockets* ou TLI, por exemplo, aplicações de usuários criam processos (remotos) através de primitivas como *rexec( )*. Neste caso, a máquina destino precisa ser especificada, sendo considerado para execução o ambiente da máquina remota, incluindo o arquivo executável de dados, *path*, etc. A chamada é pouco prática, sendo usualmente necessário fornecer a senha do usuário durante a execução da mesma. Para acompanhar o andamento de cada processo remoto, é necessário abrir sessões remotas nas máquinas onde os processos são criados, para então utilizar os comandos locais tradicionais do *Unix*, como *ps* e *kill*. Para que dois processos remotos que comunicam-se através de *sockets* se sincronizem é necessário utilizar um protocolo especial com mensagens [BAA 95a].

Com o HetNOS, processos podem ser criados, listados, sincronizados e removidos através de operações transparentes de localidade. O comportamento de tais operações é similar às equivalentes *Unix*. Quando um processo é criado em um nodo remoto, diferente daquele em que o processo pai reside, o processo filho herda atributos como *tty* associada, UID (identificador de usuário), GID (identificador de grupo) e variáveis de ambiente (como *path*) [BAA 93].

Isto permite que comandos e ferramentas do HetNOS, bem como demais aplicações de usuário, utilizem serviços de gerência distribuída de processos. A influência na interface de comandos é significativa, pois um usuário pode disparar um processo remoto, acompanhar seu andamento, abortá-lo, etc., sem precisar saber em qual nodo o mesmo estava sendo executado. A filosofia utilizada é não esconder a localização dos processos, e sim evitar que o usuário precise conhecê-la [BAA 93].

Outra diferença essencial nas operações com processos no HetNOS é que todas as primitivas trabalham com argumentos que são símbolos, isto, é, elementos textuais identificadores de processos. Todas as primitivas que aceitam como argumento um ou mais nomes de processos recebem uma cadeia de caracteres. Uma vantagem deste

esquema é que as primitivas que atuam sobre um único processo podem ser transparentemente estendidas para trabalharem com múltiplos processos, sem qualquer alteração a nível de interface. Atuar sobre um único processo é um caso particular de funções que atuam sobre diversos processos [BAA 95a].

O HetNOS estende, na medida do possível, a gerência local de processos realizada pelo *Unix* para um esquema distribuído. Quando uma aplicação de usuário solicita ao HetNOS a criação de um processo, a operação é executada parte pelo núcleo do HetNOS e parte pelo *Unix*. Em outras palavras, o usuário solicita a criação de um processo ao HetNOS, que por sua vez, solicitará ao *Unix* da mesma máquina ou de outra, a criação de um processo. Estruturas internas serão atualizadas no núcleo do HetNOS, pois este controla uma série de atributos (*Unix* ou não) de processos. A hierarquia de processos *Unix*, que é local, é substituída por uma nova hierarquia global HetNOS [BAA 95a] [WAL 84].

Uma diferença importante da gerência *Unix* para o HetNOS é que neste último a morte de um processo provoca a morte de todos os processos filhos derivados deste [BAA 95a] [WAL 84].

4.3.1 Esquema de Sessões

Para que uma aplicação possa acessar os serviços do HetNOS ela precisa primeiro fazer parte de uma sessão de usuário. Se o processo foi criado através de uma primitiva HetNOS, o processo pai já fazia parte de uma sessão e o filho faz parte da mesma sessão (segundo uma árvore genealógica). Para que um processo abra uma sessão HetNOS, basta executar a chamada *h_login( )*, fornecendo na mesma o *username* e a senha HetNOS, mais o nome a ser atribuído ao processo inicial (de login). Todos os processos criados a partir desse processo inicial farão parte dessa mesma sessão, incluindo-se aí os processos remotos [BAA 95a].

A função *h_login( )* retorna o nome atribuído ao processo, que será diferente do primeiro argumento apenas se ele for "ANY", ao que o HetNOS escolherá um nome para o processo. Este nome pode ser contido de outras duas formas: através da variável global *whois* e da função *h_get_name( )*, a seguir discutida.

TABELA 4.6 - Primitivas de Controle de Sessão [BAA 95a]

`char *h_login(char *nome_login, char *user, char *senha)` abre uma sessão de trabalho
`int h_logout()` fecha uma sessão de trabalho

Para terminar uma sessão, utiliza-se a primitiva *h_logout( )*. Esta função indica ao HetNOS que elimine todos os processos filhos, feche arquivos abertos, encerre conexões, etc. Entretanto, o processo que realizou a chamada *h_login( )* continua existindo no *Unix*, o programa aciona *exit( )* quando achar necessário. A Tabela 4.6 apresenta as primitivas de controle de sessão atualmente disponíveis [BAA 95a].

4.3.2 Requisição de Informações sobre Processos

O HetNOS oferece um conjunto de primitivas para obtenção de informações sobre seus processos. As funções mais simples retornam o nome próprio do processo ou do pai em uma área alocada dinamicamente pela própria rotina, não sendo necessário fornecer um *buffer* para armazenamento da resposta. Há funções para localização de um processo na rede, bem como para obtenção de atributos sobre o mesmo. A Tabela 4.7 apresenta as primitivas atualmente disponíveis [BAA 95a].

TABELA 4.7 - Primitivas para Obtenção de Informações sobre Processos [BAA 95a]

`char *h_get_name()` retorna uma cadeia contendo o nome do próprio processo (idêntico ao conteúdo de *whois*)
`char *h_get_parent_name()` retorna uma cadeia contendo o nome do processo pai
`char *h_find_proc()` retorna uma cadeia contendo o nome do nodo onde reside determinado processo

4.3.3 Criação de Processos

A criação de processos no HetNOS pode ocorrer de duas formas distintas, a saber: criação de processos por duplicação local e criação de processos por duplicação e execução.

4.3.3.1 Criação de Processos por Duplicação Local

No *Unix* a criação de processos é obtida através da chamada *fork( )*. Um processo é criado através da "duplicação", ou seja, a imagem do processo pai é duplicada gerando o processo filho. Uma nova entrada na tabela de processos é criada. Pai e filho executam o mesmo código (provavelmente compartilhado), mas possuem uma cópia individual (inicialmente idêntica) dos dados. Registradores são inicialmente os mesmos, à excessão do contador de programa e daqueles registradores que se

referem à posições absolutas de memória. Como os descritores são copiados do pai para o filho, este recebe abertos todos os arquivos (e conexões) mantidas abertas pelo pai no momento da criação [STE 90].

O HetNOS possui uma primitiva que "imita" a operação *fork* do *Unix*. A primitiva *Unix* não pode ser diretamente utilizada por uma aplicação de usuário porque a gerência de processos é parcialmente delegada ao HetNOS. Este precisa verificar a unicidade do nome da aplicação, realizar contabilizações, atualizar tabelas internas, etc. antes de providenciar o *fork* da aplicação. Para o usuário, o resultado final é o mesmo. O HetNOS, como sistema de rede não tem como realizar um *fork* distribuído, criando um processo duplicado em outra máquina [BAA 95a].

A chamada aceita um argumento (diferentemente da função do *Unix*), que consiste no nome do processo a ser criado. É possível que o usuário delegue ao HetNOS a escolha do nome para seu processo filho, bastando fornecer "ANY" como parâmetro. O nome escolhido é o valor de retorno da função para o processo pai, em caso de erro. Para o processo filho, sempre é retornado *NULL*. Para distinguir o processo filho de um erro, é necessário testar o valor da variável *h_error*. A Tabela 4.8 apresenta a primitiva para duplicação atualmente disponível [BAA 95a].

TABELA 4.8 - Primitiva para Criação de Processo por Duplicação [BAA 95a]

`char *h_fork(char *nome_filho)` duplicação de processo, batizando o novo processo de acordo com `nome_filho`

4.3.3.2 Criação de Processos por Duplicação e Execução

No sistema *Unix*, para que se execute um programa qualquer deve-se primeiramente duplicar um processo (com *fork( )*) para então executar o programa em questão (com *exec( )* ou uma de suas variantes). A função *exec( )* carrega um novo executável. Os sistemas *Unix* geralmente oferecem ainda a função *rexec( )* [STE 90].

TABELA 4.9 - Primitivas para Criação de Processos por Duplicação/Execução [BAA 95a]

`char *h_loc_exec(char *nome_filho, char *comando)` duplicação do processo e execução de um comando local
`char *h_rem_exec(char *nome_filho, char *host, char *comando)` duplicação do processo e execução de um comando remoto
`char *h_exec(char *nome_filho, char *comando)` duplicação do processo e execução de um comando no melhor nodo

O HetNOS não possui uma primitiva semelhante ao *exec( )*. Em compensação, existem três primitivas semelhantes ao *rexec( )* do *Unix*. As variações da primitiva estão relacionadas ao local da execução. Todas elas retornam o nome do processo criado em caso de sucesso (útil no caso em que "ANY" é passado como nome de processo). A Tabela 4.9 ilustra as primitivas disponíveis [BAA 95a].

4.3.4 Inicialização e Término de Processo

Um novo processo é criado por uma das primitivas para criação de processos apresentadas anteriormente. O novo processo é reconhecido como "apto para processamento" apenas quando executa a chamada de inicialização *h_init( )*. Enquanto isso, o processo requisitante (pai) fica bloqueado esperando o resultado da operação. Esta chamada inicializa tabelas internas no processo, sendo o nome do processo carregado em uma variável denominada *whois*, pertencente à biblioteca. Esta variável pode e deve ser utilizada por aplicações [BAA 95a].

Há duas funções complementares ao *h_init( )*: *h_terminate( )* e *h_exit( )*. Um processo deixa de existir para o HetNOS (mas pode continuar executando pelo *Unix*) assim que executa a chamada *h_terminate( )*. Se o processo desejar acabar também no *Unix* (caso mais comum), então ele pode executar a função *h_exit( )*, que também encerra o processo no *Unix*. Com esta operação é possível passar uma *string* de *status* ao processo pai, que pode recebê-la com *h_wait( )*. A tabela 4.10 ilustra as primitivas disponíveis [BAA 95a].

TABELA 4.10 - Primitivas para Inicialização e Término de Processo [BAA 95a]

`int h_init()` inicializa estruturas internas - configura comunicação com o HetNOS
`int h_terminate()` finaliza a utilização dos serviços HetNOS (aplicações que utilizam `h_login()` devem acionar `h_logout()`)
`int h_exit(char *text)` finaliza a utilização dos serviços HetNOS e *Unix*, retornando uma cadeia de caracteres ao processo pai (aplicações que utilizam `h_login()` devem acionar `h_logout()`)

4.3.5 Eliminação de Processo

Processos podem ser eliminados (abortados) através da primitiva *h_kill( )*. Apesar do nome sugerir um comportamento idêntico ao *kill( )* do *Unix*, tal não se verifica. Em primeiro lugar, o *kill( )* do *Unix* serve para enviar um sinal a um processo

[STE 90], enquanto o *h_kill( )* do HetNOS serve para eliminar processos. Isto porque o HetNOS ainda não suporta a utilização de sinais tal como no *Unix*. Em segundo lugar, quando um processo morre todos os filhos e derivados são também eliminados. Por fim, *h_kill( )* aceita como argumento de entrada uma lista de processos a serem eliminados, enquanto a chamada *kill( )* atua sobre apenas um processo de cada vez [BAA 95a]. A tabela 4.11 apresenta a primitiva para eliminação de processos.

TABELA 4.11 - Primitiva para Eliminação de Processos [BAA 95a]

`int h_kill(char *lista_processos)` aborta a execução de um determinado processo

4.3.6 Sincronização entre Processos

Para que um processo espere o término de outro, basta executar a função *h_wait( )*. Esta função espera que o processo termine, retornando uma cadeia de caracteres informada como *status* no momento do *h_exit( )*. Caso o processo tenha terminado com *h_terminate( )*, a cadeia apontada é vazia. Em caso de erro, *NULL* é retornado. Existem outras formas de sincronizaçãom através de mensagens síncronas ou de primitivas de sincronização de mensagens [BAA 95a]. A Tabela 4.12 apresenta a primitiva para sincronização entre processos.

TABELA 4.12 - Primitiva para Sincronização entre Processos [BAA 95a]

`char *h_wait(char *nome_proc)` espera pelo término de um outro processo, local ou remoto

4.3.7 Primitivas para Exame de Fila de Mensagens

O HetNOS permite que um processo examine a sua própria fila de mensagens ou a de outro processo. A primitiva mais simples apenas conta o número de mensagens presentes na final do próprio processo de acordo com a especificação de um origem (que pode ser "ANY", por exemplo). A outra primitiva é mais poderosa, na medida que permite investigar a fila de mensagens de outro processo (ainda segundo uma especificação de origem). A resposta retornada nesse caso é o conteúdo inicial de cada mensagem, sendo que cada mensagem é iniciada pelo nome do origem da mensagem, seguida de um sinal de dois pontos e um espaço ("origem: msg ...") [BAA 95a].

O HetNOS oferece uma primitiva de sincronização que atua sobre a fila de mensagens de um determinado processo. A primitiva *h_sync_send( )* equivale à primitiva *h_send( )* seguida de um *h_wait_recv( )*. Esta função faz com que o processo que a chamou fique bloqueado até o momento em que todas as mensagens forem lidas [BAA 95a].

As primitivas para gerência da fila de mensagens são apresentadas na Tabela 4.13 [BAA 95a].

TABELA 4.13 - Primitivas para Gerência de Fila de Mensagens [BAA 95a]

`int h_count_messages(char *origem)` conta quantas mensagens estão disponíveis na fila do processo, de acordo com a especificação do processo `origem`
`char *h_examine_queue(char *origem, char *destino)` examina a fila de mensagens do processo `destino` selecionando as mensagens de acordo com `origem`; o início de cada mensagem selecionada é retornado através de um *buffer* estático declarado na função

4.3.8 Primitivas para Exame de *Host*

O HetNOS oferece ainda primitivas para verificação da configuração corrente dos nodos da rede HetNOS. Estas primitivas são especialmente utilizadas para a realização do balanceamento de carga, visando um equilíbrio na ocupação das estações [BAA 95a]. As primitivas para exame de *host* são apresentadas na Tabela 4.14 [BAA 95a].

TABELA 4.14 - Primitivas para Exame de *Host* [BAA 95a]

`int h_get_host_load(char *nomehost, int *carga_cpu, *carga_mem)` obtém a carga de cpu e de memória de um determinado *host*
`int h_get_lighest_host(char const *nomehost, int *carga_cpu, *carga_mem)` obtém o nome, a carga de cpu e de memória do *host* que estiver menos ocupado no momento da chamada

4.4 Arquitetura do HetNOS

O HetNOS é um sistema operacional de rede completamente definido, porém não totalmente depurado. A estrutura do sistema está organizada em camadas hierárquicas, e foi influenciada pelo MINIX [TAN 87]. Cada camada pode estar dividida em dois ou mais módulos, sendo que cada módulo é implementado como um processo *Unix* executando a nível de usuário. Até o presente momento, não há memória compartilhada entre os módulos do sistema, embora se estude esse mecanismo a nível de *Unix* como forma de acelerar o paradigma de troca de mensagens. As mensagens são transmitidas através de um protocolo executado sobre o fluxo de *bytes* propiciado pelo TCP. Este esquema de comunicação permitiu a construção de um *software* modular, com desenvolvimento independente de processos servidores [BAA 95b].

A arquitetura local do HetNOS está ilustrada na figura 4.15, e corresponde às seguintes camadas ou níveis [BAA 95b]:

- nível 1 - ***Unix*, sistema operacional nativo**: funções tradicionais do *Unix*, mais comunicação em rede;
- nível 2 - **DCL, núcleo distribuído do HetNOS**: implementa comunicação entre processos, gerência de processos distribuídos, monitoração da rede (juntamente com o Servidor de Boot - BS), esquema de autorização (juntamente com o Servidor de Autorização - AS);
- nível 3 - **SS, servidores do sistema**: complementam a funcionalidade do núcleo distribuído, com serviços como espaço de tuplas distribuídas, esquema de controle de sessões, uso de recursos, transferência de mensagens tipadas, facilidades para acesso a arquivos remotos não cobertos pelo NFS, etc.;
- nível 4 - **aplicações distribuídas**: são as ferramentas de desenvolvimento do HetNOS e as aplicações distribuídas.

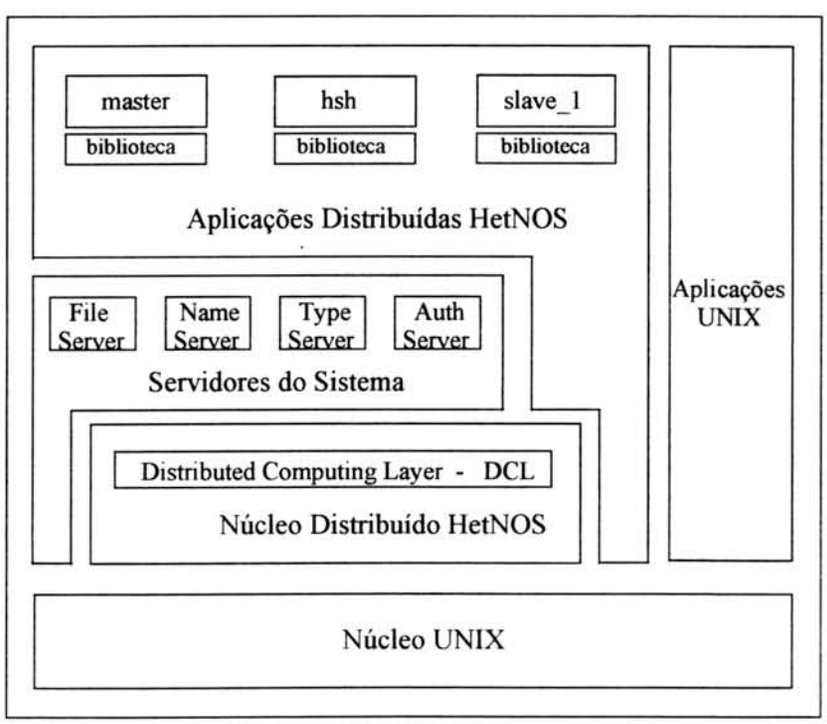


FIGURA 4.1 - Arquitetura Local do HetNOS [BAR 95c]

Cada módulo do HetNOS atua como um servidor concorrente, que recebe e responde múltiplos pedidos de forma concorrente. Não há a criação de novos processos para atendimento de pedidos de usuário. No atendimento à requisições de usuário, a

comunicação com outros módulos locais ou remotos é realizada de forma assíncrona, o que aumenta substancialmente a complexidade do código [BAA 95b].

Os módulos do HetNOS executam a nível de usuário *Unix*, e desta forma não possuem acesso às tabelas internas do *Unix*. Tais módulos possuem a mesma prioridade e restrições que outras aplicações de usuário. Por esta razão, não é possível migrar processos entre estações, ou criar um processo remoto através da duplicação de imagem.

As classes de serviços básicas oferecidas pelo HetNOS compreendem [BAA 95b]:

- operações distribuídas: configuração de rede, comunicação e sincronização entre processos;
- serviços HetNOS: servidor de nomes (*Name Server* - NS), servidor de tipos (*Type Server* - TS);
- extensões ao *Unix* tradicional: operações sobre arquivos (servidor de arquivos - *File Server* - FS), linguagem de comandos distribuída, servidor de autorização (*Auth Server* - AS) e assim por diante.

4.5 Questões de Projeto

Na fase inicial do projeto do ADC foram analisadas diversas alternativas de implementação, desde o uso de *sockets*, até o desenvolvimento do ambiente com o auxílio das bibliotecas *p4* e *PVM* (*Parallel Virtual Machine*).

A implementação do ambiente através de uma bilioteca de primitivas de rede para RPC (*Remote Procedure Call*) tal como *sockets* [AND 93], embora apresente alto desempenho, implica no conhecimento e especificação de detalhes de baixo nível, como por exemplo, endereço de máquina, número de porta, localização dos diversos processos na rede.

p4 é uma biblioteca de funções e macros desenvolvidas para máquinas paralelas [BUT 92]. Suas aplicações utilizam as linguagens C ou FORTRAN. Apesar da comunicação dos processos ser por meio de troca de mensagens, há restrições, uma vez que somente é possível a comunicação entre módulos do mesmo programa. Sendo assim, não é possível para um programa enviar uma mensagem para um programa diferente. Outra desvantagem da biblioteca *p4* é o tamanho de seus executáveis. Todos os seus arquivos executáveis contêm uma cópia do código de comunicação do sistema, o que aumenta siginicativamente o seu tamanho.

A biblioteca *PVM* [GEI 90], da mesma forma que *p4*, permite a utilização das linguagens C ou FORTRAN em suas aplicações. *PVM* apresenta a facilidade de associar um único identificador global para cada processo, embora este identificador seja do tipo inteiro (*task id*). O identificador numérico de um processo em *PVM* é dinamicamente gerado pelo sistema (identificadores não podem ser determinados em tempo de compilação).

Apesar da biblioteca *PVM* e do sistema HetNOS oferecerem níveis de desempenho equivalentes, a opção pelo sistema HetNOS teve como base a perspectiva futura de adaptação do ADC às necessidades de confiabilidade do HetNOS.

A escolha do sistema HetNOS como base para implementação do ADC tem também sua fundamentação nas facilidades proporcionadas pelo mesmo. O HetNOS apresenta um esquema simbólico de identificação de processos, com o qual os mesmos podem ser identificados de forma global e independente de localidade. Processos enviam e recebem mensagens usando sua identificação, que é única em toda a rede [BAA 93a].

O sistema HetNOS oferece a flexibilidade não encontrada no modelo RPC (*Remote Procedure Call*), uma vez que libera o usuário de tarefas de gerenciamento de troca de mensagens, as quais estão associadas a *sockets* ou TLI (*Transport Layer Interface*) [BAA 93a].

A interface do HetNOS é bastante fácil de usar e sua simplicidade decorre, primeiramente, por omitir do usuário uma série de detalhes, como por exemplo: a determinação dos endereços de rede, conhecimentos sobre a estrutura específica do protocolo, o estabelecimento de uma conexão, etc. Outro fator que torna o HetNOS simples é a substituição, sempre que possível, de códigos numéricos por nomes, e estruturas de dados por cadeias de caracteres. Dessa forma, a nível de funções e linhas de comando, usuários trabalham mais com símbolos do que com valores.

Como ambiente para desenvolvimento de aplicações distribuídas, o HetNOS utiliza um núcleo distribuído e um conjunto de módulos servidores para oferecer, através de funções de alto-nível, uma gama de serviços a programas de usuários. O desenvolvimento de tais programas são auxiliados por ferramentas de apoio, tal como um interpretador de comandos especial. Entre as vantagens propiciadas pelo ambiente estão: variedade de primitivas de comunicação e de gerência de processos, simplicidade das primitivas, transparência de localidade, ambiente de desenvolvimento, comunicaçào entre aplicações não relacionadas, inexistência de módulos extra de programa fonte a integrar (tal como esqueletos de clientes e servidores), etc [BAA 95a].

5 Descrição do Ambiente

Com o objetivo de avaliar o comportamento dos diversos processos de um grupo de difusão frente à introdução falhas, foi realizado um estudo de vários Protocolos de Difusão Confiável [BAR 94], onde foram observadas suas características, propriedades, vantagens e desvantagens. A partir deste estudo, surgiu a idéia de se projetar um ambiente capaz de analisar os protocolos de difusão sob determinados aspectos.

Apesar de não se restringir à análise de protocolos de difusão confiável, o ambiente em questão recebe a denominação de *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*. Isto se deve ao fato de que um protocolo de difusão atômica ou difusão causal faz parte da classe de protocolos de difusão confiável, uma vez que os mesmos obedecem a propriedade de confiabilidade (seção 3.2). Além destes tipos de difusão, o ambiente é capaz de analisar modelos cuja difusão é não-ordenada, bem como modelos com difusão FIFO, sendo que estes estão geralmente associados a modelos de difusão confiável (seção 3.3.3).

A execução do ADC se dá em duas etapas: experimentação e análise, conforme serão apresentadas no capítulo 6. Na primeira etapa é realizada uma experimentação a partir de um modelo previamente definido. Nesta etapa são extraídos dados através da simulação da execução do modelo proposto. Estes dados serão objeto de uma análise, a qual corresponde a segunda etapa do ADC.

O ADC executa o modelo proposto em um ambiente totalmente controlado e suposto livre de qualquer tipo de falha, exceto as introduzidas pelo próprio ADC.

A partir de um determinado conjunto de entradas, definido como modelo proposto, a função do ambiente é verificar, de modo experimental, que tipo de protocolo de difusão de mensagens melhor se adapta às exigências de alta confiabilidade, baixa complexidade, baixo custo e alto desempenho. Tais exigências são relacionadas diretamente aos dados de análise, os quais são obtidos durante a etapa de experimentação. O *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)* pretende analisar os diversos aspectos relacionados à *bufferização* de mensagens, restrição da semântica de falhas, ordenação de mensagens, determinação do consenso e tráfego de mensagens, os quais influem diretamente no desempenho, confiabilidade, complexidade e custo do modelo.

5.1 Estrutura do ADC

O *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)* é composto por seis módulos, conforme ilustra a figura 5.1 [BAR 95]. Compreendem estes módulos: Módulo Entrada de Dados, Módulo Protocolo, Módulo Externo, Módulo Introdução de Falhas, Módulo Análise e Módulo Resultados.

Cada módulo do ambiente é implementado como um processo independente. A criação dos processos no HetNOS obedece uma hierarquia, onde o processo inicial do ADC (Módulo Entrada de Dados), é criado diretamente pelo HetNOS, conforme descrito no Anexo A-1, e os demais estabelecem uma estrutura semelhante a uma árvore de processos.

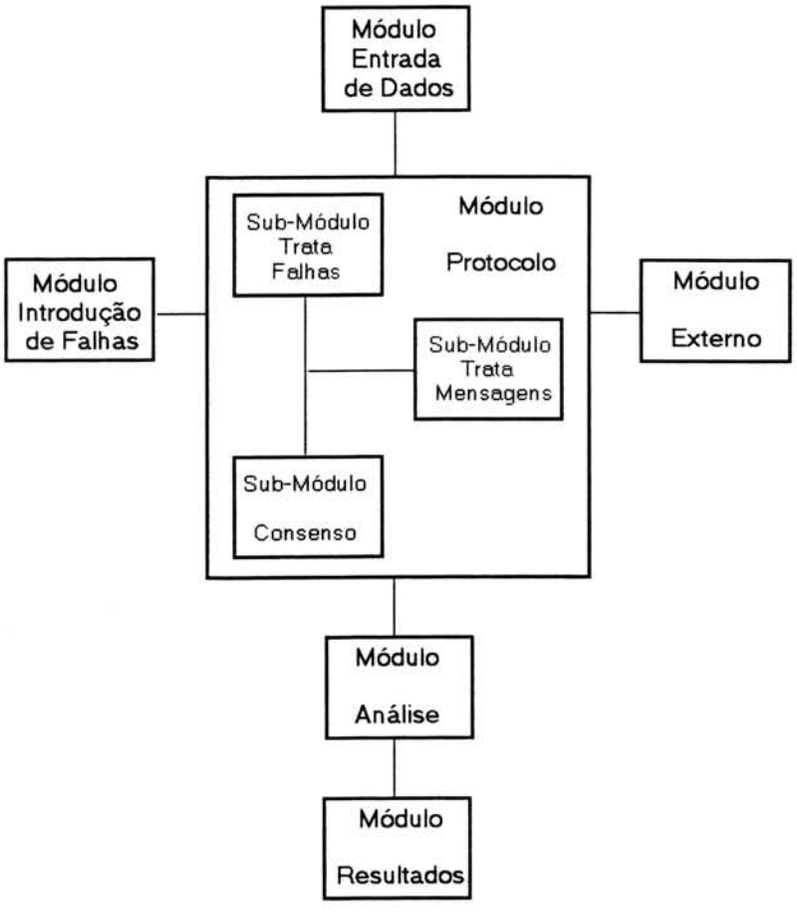


FIGURA 5.1 - Estrutura do ADC [BAR 95d]

De acordo com a característica interativa do ambiente, faz-se necessária a especificação de um modelo de sistema, sobre o qual serão realizadas análises. Este modelo, denominado modelo proposto, traz consigo um conjunto de informações que determinarão a escolha do protocolo adequado pelo ADC. Compreende este conjunto de informações o número de processos que participarão da simulação do modelo, o número de estações nas quais os processos irão executar, o tipo de terminação de protocolo e o tipo de ordenação de mensagens a serem implementados, bem como os tipos de falhas as quais o modelo deve suportar. A coleta de tais informações é realizada pelo módulo inicial do modelo, denominado Módulo Entrada de Dados.

A composição da rede se dá através das informações referentes ao número de processos que participarão da simulação do modelo e ao número de estações nas quais os processos irão executar. O balanceamento de carga, conforme descrito na seção 5.2.1, pode ser realizado a nível de usuário ou a nível de sistema operacional, uma vez que o HetNOS oferece esta facilidade.

O tipo de terminação de protocolo a ser utilizado pelo modelo envolve, conforme mencionado na seção 2, a existência ou não de um atraso máximo no tempo de entrega das mensagens. Tais tipos de terminação compreendem a terminação síncrona e a terminação assíncrona de protocolo. Aos modelos com terminação assíncrona, o ADC introduziu o conceito de *timeout*, evitando, com isso, que o sistema entre em *deadlock* devido à ocorrência de uma falha. Esse *timeout* considera um tempo de terminação razoavelmente grande, não comprometendo, portanto, a característica assíncrona do modelo.

O tipo de ordenação de mensagens, o qual será implementado no modelo proposto, vem de encontro às necessidades de sincronização de mensagens. De acordo com a seção 2.1.1, o tipo de ordenação de mensagens identifica o tipo de difusão a ser implementado, dentre os tipos descritos em 3.3. Os tipos de ordenação de mensagens oferecidos pelo ADC são: sem ordenação, ordenação FIFO, ordenação total e ordenação causal. Destes tipos de ordenação de mensagens surgem os tipos de difusão, a saber: difusão não-ordenada, difusão FIFO, difusão confiável, difusão atômica e difusão causal [JAL 94].

Os tipos de falhas a serem suportadas pelo protocolo introduzem a necessidade de um conhecimento prévio a respeito do comportamento dos processos mediante a presença de falhas. Este conhecimento teve como base o estudo realizado em [BAR 94], onde foram analisados os protocolos de difusão confiável descritos em 3.5. O ADC segue a semântica de falhas proposta por [CRI 91], onde se tem falhas bizantinas, falhas de temporização, falhas por omissão e falhas de *crash*. Tais falhas encontram-se descritas na seção 3.1.

Após a especificação do modelo proposto a ser analisado pelo ADC, o Módulo Entrada de Dados realiza uma verificação das características propostas. Esta verificação visa a formalização do modelo como objeto de análise.

A difusão de mensagens se constitui em uma das mais importantes fases no processo de experimentação. Isto porque, a partir desta fase serão iniciados os processos de extração de dados. Tais dados serão utilizados pelo ADC como objeto de análise, cujos resultados correspondem ao produto do ambiente. É no Módulo Protocolo, o qual é responsável pela simulação do modelo, que inicia a fase de difusão propriamente dita. O Módulo Protocolo interage com os demais através de um conjunto de rotinas, as quais compreendem três Sub-Módulos: Trata-Mensagens, Consenso e Trata-Falhas.

Após a composição da rede e organização da lista de processos, dá-se início a fase de difusão de mensagens ao longo da rede. É de responsabilidade do Sub-Módulo Trata-Mensagens a recepção e o tratamento de todas as mensagens que circulam pelo anel lógico HetNOS. Nesta etapa podem ser introduzidas falhas, de acordo com o que foi especificado no modelo proposto. Tais falhas são manipuladas pelo Sub-Módulo Trata-Falhas. Este tem como função a preservação da consistência do sistema a despeito das falhas inseridas pelo Módulo Introdução de Falhas.

A característica consistente dos sistemas distribuídos introduz a necessidade de uma verificação entre todos os processos sobre os valores disseminados pela rede durante a fase de difusão. Este procedimento é realizado pelo Módulo Consenso. Devido ao processo de coleta de informações sobre os valores difundidos ocorrer via troca de mensagens, é possível que durante tal processo ocorram falhas. Falhas estas inseridas pelo Módulo Introdução de Falhas. Portanto, as rotinas que compõem os três Sub-Módulos do Módulo Protocolo trabalham em conjunto constantemente durante a experimentação do modelo.

O Módulo Externo, executado pelo processo coordenador, tem como função o gerenciamento dos demais processos durante a simulação do modelo proposto. É também encargo do Módulo Externo a retransmissão de mensagens perdidas ou não entregues ao devido destino, quando solicitada por um processo do grupo de difusão.

O Módulo Introdução de Falhas, como o próprio nome sugere, é responsável pela geração e introdução de falhas no sistema, de acordo com a especificação proposta pelo modelo no Módulo Entrada de Dados. A introdução das falhas no ADC é realizada por meio de mensagens. Conforme mencionado anteriormente, cabe ao Sub-Módulo Trata-Falhas, o qual pertence ao Módulo Protocolo, a manipulação das falhas geradas pelo Módulo Introdução de Falhas.

O Módulo Análise é o responsável pela interpretação das informações recebidas dos outros módulos do ADC. Através do procedimento de interpretação, é estabelecido o relacionamento dos dados obtidos com a experimentação com os valores suportados pelo modelo.

Após a avaliação do modelo proposto, realizada pelo Módulo Análise, conforme os parâmetros descritos anteriormente, são exibidas as conclusões as quais se chegou em relação à experimentação do modelo. Esta função pertence ao Módulo Resultados.

5.2 Descrição dos Módulos do ADC

A seguir são descritos os seis módulos pertencentes ao ADC, bem como os relacionamentos entre os mesmos. A execução do ADC é apresentada no capítulo 6, onde serão vistos os detalhes de implementação que desta seção foram abstraídos.

5.2.1 Módulo Entrada de Dados

É o módulo inicial do ambiente (figura 5.2) sendo, por definição, criado diretamente pelo HetNOS através da linha de comando, conforme apresentado no Anexo A-1.

FIGURA 5.2 - Módulo Entrada de Dados

O Módulo Entrada de Dados é o módulo responsável pela aquisição do modelo proposto. Modelo proposto é o nome dado ao conjunto de informações que caracterizam a especificação a ser analisada. A utilização do ADC está diretamente vinculada ao conhecimento das necessidades e restrições de um protocolo de difusão confiável. A partir deste conhecimento, torna-se possível a formulação do modelo proposto. Este conhecimento envolve desde aspectos de terminação do protocolo, ordenação de mensagens e semântica de falhas até a avaliação dos resultados obtidos com a experimentação e análise. Tais resultados são exibidos como produto da experimentação, isto é, da simulação do modelo proposto.

Com isso, faz-se necessária a formulação do modelo proposto. O conjunto de informações referentes ao modelo proposto compreende: o número de processos que participarão da simulação do modelo, o número de estações nas quais os processos irão executar, o tipo de terminação de protocolo a ser implementado, o tipo de ordenação de mensagens a ser implementado e os tipos de falhas a serem suportadas na simulação do modelo proposto. A descrição destas informações é apresentada a seguir:

a) *número de processos que participarão da simulação do modelo*

Este número indica ao ADC quantos processos devem ser requisitados ao HetNOS quando da etapa de criação de processos. É importante salientar que existem duas etapas de criação de processos: a criação dos módulos do ADC e a criação dos processos que executarão sobre os módulos do ADC. A primeira etapa de criação de processos constitui a formação da aplicação distribuída HetNOS, isto é, do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*. A formação do ADC consiste na criação dos seus seis módulos, a saber: Módulo Entrada

de dados, Módulo Protocolo, Módulo Externo, Módulo Introdução e Falhas, Módulo Análise e Módulo Resultados.

A segunda etapa de criação de processos corresponde à etapa de criação dos processos componentes do modelo proposto, ou seja, são os processos que irão rodar sobre os módulos do ADC. Por convenção, adotou-se a seguinte política na criação dos processos: os processos correspondentes aos módulos do ADC (primeira etapa de criação de processos) são criados de forma transparente ao usuário. Já os processos da segunda etapa de criação podem ser criados em estações pré-determinadas, seguindo, no entanto, a metodologia para balanceamento de carga utilizada pelo HetNOS, conforme será descrito no item a seguir.

b) *número de estações nas quais o modelo proposto irá executar*

Este número é dependente do número de estações nas quais o HetNOS foi ativado, conforme apresentado no Anexo A-1: Ativação do HetNOS. É desnecessário observar que o número de estações nas quais o protocolo irá executar deve ser menor ou igual ao número total de estações nas quais o HetNOS encontra-se ativo.

A partir deste número, juntamente com o número de processos que participarão da simulação do modelo, é possível realizar o que se chama de balanceamento de carga, visando, desta forma, o equilíbrio na ocupação das estações. No HetNOS, o balanceamento de carga é caracterizado como estático, sendo somente possível de ser realizado no momento da criação de processos. Isto porque o HetNOS não oferece os recursos necessários para a migração de processos, conforme descrito em 4.4. A tarefa de balanceamento de carga no HetNOS pode ser realizada a nível de usuário ou a nível de sistema operacional. Para o balanceamento de carga a nível de usuário, o HetNOS oferece primitivas para auxiliá-lo nesta tarefa (apresentadas na seção 4.3.8). No balanceamento de carga a nível de sistema opreacional, o próprio HetNOS se encarrega de distribuir os processos sobre a rede.

c) *tipo de terminação de protocolo a ser implementado*

Esta informação compreende os tipos básicos de terminação de protocolo existentes: terminação assíncrona ou terminação síncrona. A razão pela qual a utilização do ADC está diretamente vinculada ao conhecimento das necessidades e restrições de um protocolo de difusão confiável é a determinação tanto desta informação quanto das informações subsequentes. Cabe ressaltar que, na maioria das aplicações, é de fundamental importância o estabelecimento de limites temporais na entrega de mensagens.

Por questões de implementação, aos modelos cuja opção de terminação de mensagens é assíncrona, o ADC introduziu o conceito de *timeout*. Isto porque a não determinação de uma limitação no tempo de entrega das mensagens poderia conduzir o modelo a um estado de *deadlock*. Entretanto, a introdução de um *timeout*, o qual considera um tempo de terminação razoavelmente grande, resolve o problema de um

provável estado de *deadlock*, não comprometendo, contudo, a característica assíncrona do modelo.

d) *tipo de ordenação de mensagens a ser implementado*

O modelo proposto deve trazer consigo informações sobre que tipo de ordenação de mensagens deve ser aplicado, dentre os tipos apresentados na seção 2.1.1: sem ordenação, ordenação FIFO, ordenação total e ordenação causal. Os tipos de ordenação de mensagens são decisivos na determinação do tipo de difusão de mensagens a ser implementado no modelo (descritos na seção 3.3), implicando, consequentemente, no desempenho e confiabilidade do mesmo. Os tipos de difusão de mensagens que implementam os tipos de ordenação acima mencionados são: difusão não-ordenada, difusão FIFO, difusão confiável, difusão atômica e difusão causal. Esta classificação de difusão, conforme mencionado anteriormente, segue a proposta de [JAL 94].

e) *tipos de falhas a serem suportadas na simulação do modelo proposto*

A semântica de falhas adotada pelo ADC está de acordo com a classificação de [CRI 91] exibida na seção 2.1, a qual apresenta falhas bizantinas, falhas de temporização, falhas por omissão e falhas de *crash*. As falhas bizantinas (ou maliciosas, ou ainda arbitrárias) envolvem falhas de valor e falhas de temporização. Estas últimas, por sua vez, podem ocorrer por atraso ou por avanço. Por questões de implementação, o ADC somente suporta falhas de temporização por atraso. Um caso particular de falhas de temporização são as falhas de omissão, as quais envolvem as falhas com maior grau de restrição: as falhas de *crash*.

Conforme descrito anteriormente, a determinação desta informação exige um conhecimento prévio sobre protocolos de difusão confiável, uma vez que, por exemplo, com falhas do tipo bizantinas e terminação assíncrona foi comprovada em [FIS 85] a impossibilidade de alcançar consenso. Um outro exemplo que demonstra total desconhecimento no assunto é a suposição de um modelo com terminação assíncrona que suporte falhas de temporização.

Por definição, o ADC considera a suposição de [CRI 91] no que diz respeito aos tipos de falhas citados anteriormente. Segundo [CRI 91], esta semântica de falhas forma uma hierarquia de severidade crescente, onde as falhas de *crash* estão contidas nas falhas por omissão, que estão contidas nas falhas de temporização, que por sua vez estão contidas nas falhas bizantinas. Tal citação pode ser representada por:

falhas de *crash* $\subset$ falhas por omissão $\subset$ falhas de temporização $\subset$ falhas bizantinas

Outro fato a ser considerado quanto aos tipos de falhas diz respeito a sua restrição: conforme [CRI 91] quanto menos restritiva a semântica de falhas de um modelo, ou seja, quanto mais robusto o modelo, mais cara e complexa torna-se sua

implementação. Sendo assim, os tipos de falhas assumidas quando um modelo é proposto são determinantes para o custo e o desempenho do mesmo [BAR 94].

Após a especificado o modelo, o qual será objeto da experimentação e análise, o Módulo Entrada de Dados realiza o que se pode chamar de verificação das características propostas. Esta verificação consiste na avaliação do modelo proposto frente às necessidades e restrições de implementação impostas pelo ADC. Isto devido ao ADC, apesar de ser um ambiente de experimentação genérico, ter sido implementado para suportar determinadas classes de modelos. Obviamente porque seria uma tarefa bastante difícil implementar um ambiente para experimentação de todo e qualquer modelo de protocolo de difusão confiável.

Estando o modelo proposto de acordo com as exigências de implementação do ADC, o Módulo Entrada de Dados requisita ao HetNOS a criação do módulo seguinte, o Módulo Protocolo.

A criação do Módulo Protocolo é feita através da primitiva de criação por duplicação e execução oferecida pelo HetNOS *h_exec( )*, descrita em 4.3.3.2. De acordo com [BAA 95a], esta primitiva cria um processo de forma semelhante ao *rexec( )* do *Unix*. Com isso, é estabelecida uma hierarquia entre processo pai (Módulo Entrada de Dados) e processo filho (Módulo Protocolo). A figura 5.3 ilustra este procedimento, considerando a estrutura apresentada na figura 5.1.

5.2.2 Módulo Protocolo

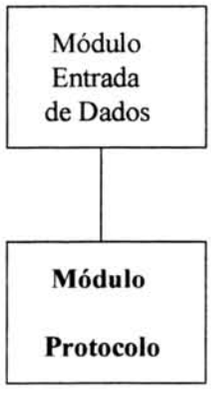


FIGURA 5.3 - Módulo Protocolo

O Módulo Protocolo, ao ser criado pelo Módulo Entrada de Dados (figura 5.3), recebe como parâmetro o conjunto de informações que compõe o modelo proposto, a saber: o número de processos que participarão da simulação do modelo, o número de

estações nas quais os processos irão executar, os tipos de terminação de protocolo e ordenação de mensagens a serem implementados e os tipos de falhas a serem suportadas na simulação do modelo proposto.

Com as informações quanto ao número de processos que farão parte da simulação do modelo e ao número de estações nas quais estes processos irão executar, o Módulo Protocolo faz a composição da rede. Por composição da rede entende-se a criação e distribuição dos processos correspondentes sobre a rede HetNOS, que por definição é em forma de anel. O procedimento de composição da rede implica em balanceamento de carga. O HetNOS oferece tanto primitivas que permitem que o usuário realize o balanceamento de carga, como a possibilidade de executar esta tarefa a nível de sistema. Conforme mencionado anteriormente, a tarefa de balanceamento de carga somente é possível ser realizada no momento da criação de processos, sendo então caracterizado como balanceamento de carga estático. Isto se deve ao fato de o HetNOS não possuir primitivas que possibilitem a migração dos processos (seção 4.4).

A implementação do ADC exige a eleição de um coordenador para o conjunto de processos que participarão da difusão. O coordenador, como o próprio nome sugere, é o responsável pela coordenação e gerenciamento de execução dos demais processos. As atribuições do coordenador encontram-se descritas na seção 5.2.3. Um coordenador é eleito a cada simulação do modelo proposto. Em caso de falha do coordenador no momento da simulação do modelo, um novo coordenador deve ser eleito.

Eleito o coordenador e designadas as estações nas quais os processos devem rodar, deve-se executar os procedimentos para a criação dos mesmos. A criação dos processos que participarão da simulação do modelo se dá através de primitivas para criação de processos por duplicação e execução (apresentadas na seção 4.3.3.2). O HetNOS fornece três primitivas para criação de processos por duplicação e execução. A determinação de quais primitivas devem ser usadas está relacionada ao local da execução de cada processo. Portanto, a criação dos processos é dependente do procedimento de balanceamento de carga, o qual define se o processos irão rodar local ou remotamente.

Como a primitiva esolhida para a criação dos processos também é responsável pela execução dos mesmos, faz-se necessário a indicação de qual rotina (arquivo com extensão .c) cada processo deve executar.

O processo coordenador executa um conjunto de rotinas especiais, as quais controlam a simulação da execução de todo o modelo. O conjunto destas rotinas forma o Módulo Externo. Portanto, a criação do processo coordenador implica na criação de mais um módulo do ADC, o Módulo Externo, conforme ilustra a figura 5.4. Na seção 5.2.3 é descrito o Módulo Externo. Os demais processos executam as mesmas rotinas, que estão sob o comando do coordenador.

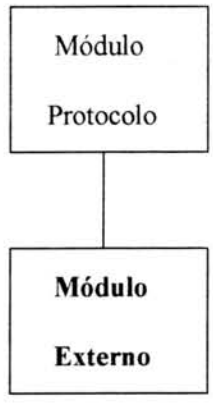


FIGURA 5.4 - Módulo Externo

O processo coordenador atua virtualmente como um processo externo ao grupo de difusão. Isto porque as atribuições do processo coordenador são totalmente diferentes das atribuições dos demais processos. Estes, por sua vez, não formam nenhum módulo do ADC, uma vez que executam rotinas pertencentes ao próprio Módulo Protocolo. Tais processos formam o que se chama *grupo de difusão*. A figura 5.5 ilustra o grupo de difusão e seu correspondente coordenador.

O Módulo Protocolo é composto por diversas rotinas, entre as quais encontram-se as rotinas de tratamento de mensagens, as rotinas de tratamento de falhas e as rotinas de determinação do consenso. Estes três conjuntos de rotinas são classificados como Sub-Módulos por fazerem parte dos procedimentos necessários para a simulação do modelo proposto. Os três Sub-Módulos do Módulo Protocolo estão descritos nos ítens 5.2.2.1, 5.2.2.2, 5.2.2.3.

Os Sub-Módulos do Módulo Protocolo são rotinas de apoio aos processos. Tanto os processos do grupo de difusão, como o coordenador interagem com estas rotinas. As rotinas pertencentes ao Sub-Módulo Trata-Mensagens estabelecem a padronização tanto no envio como na recepção das mensagens.

Para fins de simulação da execução do modelo proposto, o Módulo Protocolo cria uma lista, denominada *proc_list*, cujo conteúdo abrange todos os processos que farão parte da fase de difusão, conforme ilustra a figura 5.5. Esta lista é importante, pois é através dela que será controlada tanto a fase difusão de mensagens, como a fase de determinação do consenso.

Após criados tanto o coordenador como os demais processos, a lista contendo o nome de todos os processos (*proc_list*) é passada como parâmetro para os demais módulos no momento da criação dos mesmos. Isto porque todos os módulos que se envolvem na simulação do modelo proposto devem tomar conhecimento do conjunto de processos presentes na rede HetNOS. Como o HetNOS trabalha com *strings* (seção 4.2), os processos são identificados por nomes, ao invés de números. Estes nomes

apresentam como prefixo o nome do processo pai, ou no caso de uma hierarquia mais baixa, os nomes dos processos pai de cada processo.

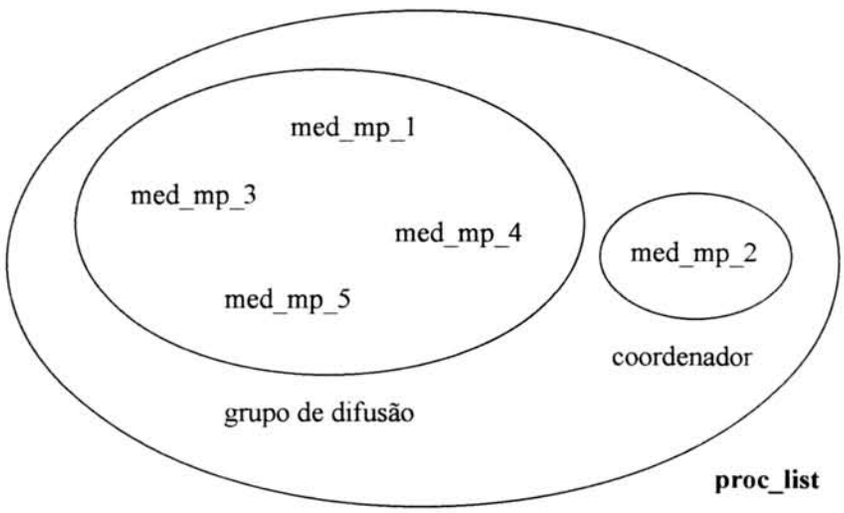


FIGURA 5.5 - Visão de uma Lista de Processos (proc_list)

Por determinação do ADC, a difusão das mensagens para o processo coordenador deve ser confiável. Isto porque o processo coordenador é o responsável pelo armazenamento das mensagens para atender a futuros pedidos de retransmissão. O modo pelo qual as mensagens são difundidas pelo ADC obedece o tipo de ordenação de mensagens exigido pelo modelo. A seção 3.3 descreve os tipos de difusão de mensagens implementados pelo ADC.

O Módulo Protocolo é responsável ainda pela criação de outros dois módulos do ADC: o Módulo Introdução de Falhas e o Módulo Análise. A criação destes módulos se faz da mesma forma que os anteriores, ou seja, através das primitivas de criação por duplicação e execução de processos (seção 4.3.3.2). No momento da criação, o Módulo Introdução de Falhas recebe como parâmetro o conjunto de informações que compõem o modelo proposto, bem como a lista de processos (*proc_list*) criada pelo Módulo Protocolo. Com isso, o Módulo Introdução de Falhas desempenha o papel de injetor de falhas do ADC. O Módulo Introdução de Falhas atua tanto nos processos do grupo de difusão como no processo coordenador. Na seção 5.2.4 encontra-se a descrição do Módulo Introdução de Falhas. A figura 5.6 ilustra o procedimento que estabelece a hierarquia entre os Módulos Protocolo e Módulo Introdução de Falhas.

O Módulo Análise, por sua vez, recebe como parâmetro o conjunto de informações referentes ao modelo proposto, bem como os dados obtidos com a simulação do mesmo. Compreendem estes dados: o número de mensagens de difusão trocadas entre os processos, o número de mensagens emitidas para a determinação do

consenso, o número de mensagens exigidas para a implementação da ordenação, o número de mensagens de difusão perdidas ou não entregues ao destino, o número de mensagens de controle (ACK's e NACK's) enviadas, o número de mensagens de controle perdidas ou não entregues ao destino, o número de mensagens de retransmissão enviadas, o número de mensagens de falha enviadas, número de mensagens armazenadas em *buffer*, o número de mensagens emitidas durante o processamento do modelo. Tais dados encontram-se descritos na seção 6.2.

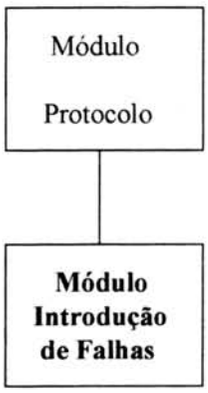


FIGURA 5.6 - Módulo Introdução de Falhas

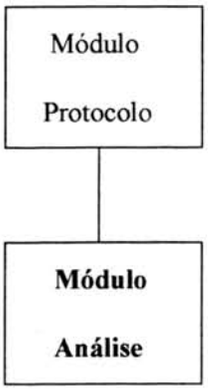


FIGURA 5.7 - Módulo Análise

O Módulo Análise é responsável, como o próprio nome diz, pela análise do modelo proposto. Esta análise é feita através da determinação das limitações do próprio modelo, de acordo com os dados obtidos com a experimentação do mesmo. Os resultados obtidos com as esta determinação são analisados segundo a confiabilidade, a complexidade, o custo e o desempenho apresentados. Como base para a implementação

da etapa de análise o ADC utiliza os resultados de protocolos de difusão confiável previamente avaliados [BAR 94]. A descrição do Módulo Análise encontra-se na seção 5.2.5, enquanto a descrição dos procedimentos relacionados à análise do modelo proposto é apresentada na seção 6.2. A figura 5.7 mostra a seqüência de criação dos módulos do ADC, onde o Módulo Protocolo dá origem ao Módulo Análise.

A seguir são descritos os Sub-Módulos componentes do Módulo Protocolo.

5.2.2.1 Sub-Módulo Trata-Mensagens

O Sub-Módulo Trata-Mensagens, através de rotinas específicas, seleciona dentre as primitivas HetNOS de envio e recepção (descritas na seções 4.2.1 e 4.2.2), as que melhor se adaptam às exigências de cada processo. Para tanto, o Sub-Módulo Trata-Mensagens apresenta um conjunto de rotinas cujo objetivo é o tratamento das mensagens recebidas. Este procedimento consiste na interpretação do tipo da mensagem, bem como o tratamento da mesma.

Visando estabelecer uma padronização, o formato de mensagens utilizado pelo ADC para envio e recepção de qualquer mensagem segue a descrição abaixo. O significado de cada parâmetro da mensagem é apresentado a seguir.

(*origem*, *timestamp*, *nro_msg*, *tipo_msg*, *conteúdo*)

- *origem*: identifica o emissor da mensagem. O ADC utiliza o conceito de mensagem identificada, devido à facilidade introduzida pelo mesmo. Isto porque, a identificação do emissor de cada mensagem possibilita a determinação de processos faltosos.
- *timestamp*: é o número de seqüência gerado pelo coordenador. O *timestamp* introduz uma seqüência global nas mensagens difundidas. Desta forma, algoritmos de ordenação de mensagens mais complexos, tais como ordenação total ou causal, podem ser facilmente simulados. Para algoritmos com ordenação causal, este parâmetro também carrega consigo a identificação de causa e efeito, a qual é introduzida por um *label*.
- *nro_msg*: corresponde a um número sequencial que cada processo adiciona às mensagens por ele enviadas. Este número é controlado pelo próprio processo, representando, assim, um número local.
- *tipo_msg*: é um código que identifica o tipo da mensagem. A utilização de uma codificação no tipo da mensagem foi introduzida por questões de implementação. Para tanto, existe uma tabela de códigos cujo conteúdo envolve todos os tipos de mensagem implementados pelo ADC, juntamente com a identificação do código. Dentre os tipos existentes, pode-se citar:

pedido de permissão para difusão, autorização para difusão e indicação de *timestamp*, pedido de retransmissão de mensagens, divulgação de novo coordenador, divulgação de nova lista de processos (*proc_list*), mensagem de falha, etc.

- *conteúdo*: relaciona-se ao tipo_msg, ou seja, dependendo do tipo de mensagem pode haver um ou mais parâmetros a serem informados. Por exemplo, se o tipo_msg for autorização para difusão e indicação de *timestamp*, o conteúdo poderia ser 5, indicando ao processo requisitante que o *timestamp* a ser anexado à mensagem deve ser 5.

A partir da exigência de um padrão pré-determinado para as mensagens, fica estabelecida uma comunicação homogênea entre todos os processos do modelo. Com isso, todo processo tem condições de identificar o emissor da mensagem, o que auxilia na identificação de processos faltosos.

A interpretação do tipo da mensagem é realizada com o auxílio da tabela de códigos cujo conteúdo apresenta todos os tipos de mensagens possíveis. Por exemplo, se o processo coordenador recebe uma mensagem de código 5, o que significa pedido de retransmissão de mensagem, o Sub-Módulo Trata-Mensagens deve verificar, através do parâmetro seguinte, o conteúdo, o número da mensagem que deve ser retransmitida. Através deste número, o processo coordenador recupera a mensagem do *buffer* e a retransmite ao requisitante, que é identificado pelo parâmetro origem.

5.2.2.2 Sub-Módulo Trata-Falhas

O Sub-Módulo Trata-Falhas, além de interagir com os processos do grupo de difusão e com o coordenador, interage com o Módulo Introdução de Falhas. O Sub-Módulo Trata-Falhas apresenta um conjunto de rotinas que monitoram a ocorrência de falhas no modelo. Falhas estas inseridas pelo Módulo Introdução de Falhas, através do conhecimento prévio das exigências do modelo. Com isso, para cada falha inserida, o Sub-Módulo Trata-Falhas utiliza um procedimento de tratamento e recuperação.

O Sub-Módulo Trata-Falhas utiliza métodos de tratamento e recuperação de falhas peculiares. A recuperação de um processo que sofre falha de *crash* é inviável, pela própria definição da falha. Isto significa que processos atingidos por falhas de *crash* não conseguem se recuperar do estado faltoso. Já os processos que sofrem falhas por omissão recuperam-se após um determinado período de tempo. Por definição, tanto os processos que sofrem falhas de *crash* como os que sofrem falhas por omissão não exibem um comportamento incorreto para o exterior.

Com falhas de temporização, a recuperação dos processos também se dá após um determinado período de tempo. Sendo que este tempo é pré-determinado pelo Módulo Introdução de Falhas. Este tipo de falha de temporização é classificado,

segundo [CRI 91], como falha de temporização por atraso. O ADC, por questões de implementação apenas oferece suporte à falhas de temporização por atraso. Isto é devido às falhas serem inseridas por meio de mensagens. Desta forma, torna-se difícil a implementação de falhas de temporização por avanço, uma vez que, neste caso, o processo deve se adiantar no recebimento de uma mensagem.

Processos que sofrem falhas bizantinas apresentam um comportamento incorreto para o exterior, devido às mensagens por eles recebidas apresentarem-se corrompidas. O algoritmo de consenso deve inibir a ação destes processos faltosos para que seja possível o alcance de uma concordância entre os processos livres de falhas.

5.2.2.3 Sub-Módulo Consenso

O Sub-Módulo Consenso, de acordo com os conceitos introduzidos na seção 3.4, tem a responsabilidade de realizar uma concordância entre os processos componentes da lista de processos (*proc_list*) livres de falhas.

Segundo o algoritmo de consenso, para cada mensagem difundida deve haver uma decisão por um valor comum entre todos os processos da lista. A maneira com que o Sub-Módulo Consenso realiza esta tarefa é requisitando diretamente ao coordenador a decisão pelo valor de difusão de cada mensagem. Por definição, assume-se que o processo coordenador sempre recebe o valor correto do emissor, ou seja, o processo coordenador não está sujeito a falhas bizantinas no momento da recepção de uma mensagem de difusão. A introdução de um valor incorreto inviabilizaria a determinação do consenso, bem como o procedimento de retransmissão de mensagens.

Com o valor de decisão para a mensagem, o Sub-Módulo Consenso divulga para todos os processos uma mensagem indicando a mensagem, a qual é objeto de concordância, e o valor decidido para a mesma. Os processos devem responder com ACK, caso concordem com o valor decidido, ou NACK, caso contrário. Com isso, o Sub-Módulo Consenso obtém a maioria dentre os processos ativos. Os processos que não respondem à requisição do Sub-Módulo são deduzidos como faltosos.

5.2.3 Módulo Externo

O Módulo Externo é um módulo criado pelo Módulo Protocolo para a ativação do processo coordenador (figura 5.4). Ao ser criado, o Módulo Externo traz consigo o conjunto de informações referentes ao modelo proposto, assim como a lista de processos (*proc_list*) com os quais vai interagir na tarefa de difusão de mensagens.

Este módulo é o responsável pelo gerenciamento e coordenação do grupo de difusão. Hierarquicamente, o Módulo Externo encontra-se localizado abaixo do Módulo

Protocolo, que por sua vez encontra-se abaixo do Módulo Entrada de Dados. Os Módulos Protocolo e Externo interagem, juntamente com o Módulo Introdução de Falhas, na simulação do modelo proposto.

A partir da escolha de um coordenador, escolha esta feita no Módulo Protocolo, o Módulo Externo apresenta um conjunto de rotinas que proporcionam ao coordenador a realização de suas funções.

O Módulo Externo é o responsável por controlar todo o fluxo de mensagens que deve percorrer a rede. Este controle é realizado com base no tipo de difusão de mensagens implementado para o modelo. O Módulo Externo, através do processo coordenador estabelece a ordenação de mensagens exigida pelo modelo.

O processo coordenador realiza todo o procedimento de armazenamento de mensagens, introduzindo com isso confiabilidade ao modelo como um todo. Esta tarefa de armazenamento de mensagens tem o propósito de atender a futuros pedidos de retransmissão. Por esta razão, o processo coordenador deve estar certo de que a mensagem de difusão por ele recebida não foi corrompida. Conforme apresentado na seção 2.2, todo *buffer* apresenta limitação de capacidade. Para contornar este problema, o processo coordenador coleta, de tempos em tempos, informações sobre quais mensagens foram recebidas por todos os processos. Com isso, o processo coordenador tem condições de descartar do *buffer* as mensagens cujo recebimento já foi acusado por todos os processos da *proc_list*.

Para a confirmação de recebimento de mensagens, o ADC utiliza o método de reconhecimento negativo ("*negative acknowledgement*") introduzido por [CHA 84]. Neste método, um processo não precisa enviar uma mensagem de reconhecimento (ACK) para cada mensagem recebida. Ao invés disso, o processo envia um "*negative ack*" quando descobre, através do sequenciamento de mensagens, que perdeu alguma mensagem. Assim, enquanto não chegam mensagens com pedidos de retransmissão ao coordenador, significa que os processos estão acompanhando a seqüência de mensagens difundidas ou, no pior caso, encontram-se em estado faltoso. Esta última hipótese é detectada quando o coordenador realiza a tarefa de coleta de informações de quais mensagens foram recebidas pelos processos para liberação do *buffer*. Tal procedimento está geralmente associado à modelos que adotam difusão total ou causal.

Falhas introduzidas no coordenador são detectadas pelos demais processos pela ausência de resposta a pedidos, tais como permissão para difusão ou retransmissão de mensagens. Uma falha do processo coordenador pode também ser detectada pelo Sub-Módulo Consenso no momento em que este requisita o valor de decisão para as mensagens. Sendo assim, os demais processos devem escolher um outro coordenador. Esta escolha é feita com base no conjunto de mensagens apresentado, ou seja, o processo que apresentar o conjunto de mensagens mais bem definido é escolhido como coordenador.

Falhas introduzidas nos processos do grupo de difusão são manipuladas pelo próprio coordenador. Este se encarrega de atualizar a lista de processos (*proc_list*) e divulgar para os demais processos e módulos do ADC. Para dar continuidade à fase de difusão, o coordenador deve estabelecer um ponto de sincronismo. A etapa de recuperação de uma falha e reinício de execução é informalmente chamada de recomposição da rede.

5.2.4 Módulo Introdução de Falhas

Este módulo, criado pelo Módulo Protocolo, conforme a figura 5.6, recebe como parâmetro o conjunto de informações correspondentes ao modelo proposto, bem como a lista de processos (*proc_list*).

O Módulo Introdução de Falhas, como o próprio nome sugere, é responsável pela geração e introdução das falhas solicitadas pelo modelo proposto. Por questões de implementação, a introdução das falhas no ADC é realizada através do envio de mensagens. São utilizadas as mesmas primitivas para envio e recepção de mensagens adotadas na comunicação entre processos e coordenador, descritas nas seções 4.2.1 e 4.2.2.

A idéia inicial era de implementar uma rotina que gerasse aleatoriamente no tempo as mensagens de falha. Infelizmente, em se tratando de sistemas distribuídos, deve-se levar em conta alguns aspectos. Tais aspectos dizem respeito ao envio e recepção de mensagens, uma vez que a determinação do momento exato que uma mensagem irá chegar ao destino constitui uma tarefa complexa em sistemas distribuídos. Isto se deve à velocidade de processamento, à carga do nodo, à carga da rede, etc. Estes aspectos, adicionados à característica de ausência de *clocks* globais, inviabilizam a utilização de passagem de tempo como sequenciamento das mensagens de falha. Com isso, adotou-se um procedimento um tanto quanto estático, mas perfeitamente adequado para os propósitos a que se destina.

O procedimento consiste na determinação lógica do momento em que a mensagem de falha deve ser recebida. Mais precisamente, se o alvo de análise forem as falhas do coordenador, serão emitidas mensagens de falha, as quais serão recebidas em pontos específicos do algoritmo, como por exemplo, no momento em que o coordenador recebe o pedido de permissão para difusão, ou no momento da retransmissão de uma mensagem para um processo requisitante. Desta forma, consegue-se gerar a falha no espaço temporal certo. O procedimento de introdução de falhas, apesar de não ser o ideal, teve que se adaptar às necessidades do ADC e às restrições impostas pelo sistema no qual o ambiente foi implementado (HetNOS).

Apesar de utilizar o mesmo formato de mensagem, o Módulo Introdução de Falhas atribui significados diferentes aos seus parâmetros. O formato padrão de mensagens adotado pelo ADC é: (*origem*, *timestamp*, *nro_msg*, *tipo_msg*, *conteúdo*). Para o Módulo Introdução de Falhas o parâmetro *origem* continua indicando o emissor da mensagem. Com isso, todo o processo que recebe uma mensagem cujo emissor é o Módulo Introdução de Falhas deduz que se trata de uma mensagem de falha.

O segundo parâmetro, o *timestamp*, carrega consigo o valor zero, isto porque esta mensagem não se trata de uma mensagem de difusão e, portanto não necessita de um número de seqüência global. O parâmetro *nro_msg*, indica, da mesma maneira que no formato utilizado para mensagens de difusão, o número de seqüência local inserido e incrementado pelo processo emissor da mensagem. O parâmetro *tipo_msg* não mais identifica o tipo da mensagem, mas o tipo de falha a ser inserida, dentre os tipos: *crash*, omissão, temporização e bizantina. Isto porque, conforme mencionado acima, o processo que recebe uma mensagem com este formato deduz que se trata de uma mensagem de falha pela identificação do emissor, que deve ser o Módulo Introdução de Falhas.

O último parâmetro tem o mesmo significado apresentado em mensagens de difusão, ou seja, o parâmetro *conteúdo* é uma complementação para o *tipo_msg*. Por exemplo, se for enviada uma mensagem de falha para um processo qualquer, indicando como *tipo_msg* o valor 3 e conteúdo o valor 18, significa, para o processo receptor, que o tipo de falha representa o código 3, dado pelo parâmetro *tipo_msg* e o conteúdo indica o valor 18. De acordo com a tabela de códigos de falha, o código 3 corresponde a falha de temporização por atraso. Portanto, o processo receptor deve reconhecer que seu processamento deve ser atraso em 18 segundos. Isto porque o valor 18, apresentado pelo parâmetro conteúdo, indica o parâmetro complementar ao parâmetro *tipo_msg*. A tabela com os códigos de falha deve ser manipulada pelo Sub-Módulo Trata-Falhas.

5.2.5 Módulo Análise

O Módulo Análise é criado pelo Módulo Protocolo, conforme mostra a figura 5.7. Ao ser criado, o Módulo Análise recebe como parâmetro o conjunto de informações que constituem o modelo proposto, assim como os dados obtidos com a experimentação, ou seja, com a simulação da execução do modelo proposto. Estes dados encontram-se descritos na seção 6.2 (ver tabela 6.1).

Realizada a etapa e experimentação, deve-se fazer a análise dos dados obtidos. A análise é realizada com base nas exigências dos sistemas distribuídos. Desta forma, o procedimento de análise é realizado com base no tráfego de mensagens, *bufferização* e ordenação de mensagens e determinação de consenso. A análise de tais parâmetros é realizada segundo às exigências de alta confiabilidade, alto desempenho, baixo custo e baixa complexidade.

A descrição detalhada das etapas de experimentação e análise é apresentada no capítulo 6, bem como uma simulação de execução de um modelo.

O Módulo Análise cria, também através das primitivas de criação por duplicação e execução de processos, o último módulo do ADC, o Módulo Resultados (figura 5.8).

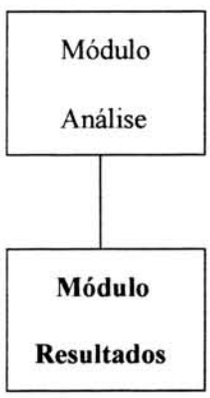


FIGURA 5.8 - Módulo Resultados

5.2.6 Módulo Resultados

O Módulo Resultados, criado pelo Módulo Análise (figura 5.8), é o módulo responsável pela apresentação dos resultados obtidos com a experimentação e análise do modelo proposto.

O objetivo do Módulo Resultados é exibir, de forma clara e objetiva, os resultados provenientes das etapas de experimentação e análise sobre um modelo para difusão confiável.

O Módulo Resultados apresenta primeiramente os dados obtidos com a experimentação do modelo proposto. Em seguida são exibidos os resultados obtidos com a análise do mesmo. A seção 6.2 apresenta uma descrição detalhada dos resultados exibidos pelo Modulo Resultados. Na seção 6.3 é descrita a execução de um experimento, juntamente com a análise do mesmo.

6 Execução do ADC

A implementação do ADC se baseia nas características e propriedades dos protocolos estudados em [BAR 94], isto é, idéias referentes aos protocolos propostos por [CHA 84], [BIR 87], [BAB 87] e [CRI 85], assim como nas características apresentadas por [GAR 82], [FIS 85], [KAA 89], [BIR 91], [COU 94], [JAL 94] e [SIN 94].

O ADC adota três abordagens, oriundas das características anteriormente mencionadas. A primeira abordagem diz respeito à centralização. Esta abordagem fundamenta-se no conceito de processo coordenador. Para um determinado grupo de processos, deve haver um processo coordenador. Embora pareça estranho utilizar uma abordagem centralizada em um sistema distribuído, esta decisão tem origem tanto nas idéias aproveitadas pelos autores acima citados, como na exigência advinda de alguns modelos. O objetivo principal na utilização desta abordagem se deve à facilidade proporcionada pelo coordenador na ordenação total de mensagens e na determinação do consenso.

Das referências mencionadas, as quais serviram de apoio no desenvolvimento do ADC, [CHA 84], [BIR 87], [BIR 93] utilizam uma abordagem centralizada de forma semelhante ao ADC.

A segunda abordagem consiste na utilização do esquema de reconhecimento negativo ("*negative acknowledgement*"). Com o esquema de reconhecimento negativo, um processo não precisa enviar uma mensagem de reconhecimento (ACK) para cada mensagem recebida. Ao invés disso, o processo envia um "*negative ack*", isto é, uma mensagem de reconhecimento negativo, caso ele descubra que perdeu uma mensagem. Tal descoberta é feita com base no sequenciamento dos *timestamps* das mensagens. A utilização desta abordagem tem fundamentação principalmente nas idéias de [CHA 84], [JAL 94].

A terceira e última abordagem utilizada na implementação do ADC não apresenta uma base específica. Esta abordagem está relacionada ao algoritmo de consenso. Apesar de ainda não ser o ideal, optou-se pela implementação de consenso centralizado. Tal opção é justificada tanto pela dificuldade de manipulação de um consenso distribuído, como pela facilidade introduzida pelo coordenador como ponto de centralização. A metodologia adotada na implementação do consenso centralizado do ADC consiste na atribuição da tarefa de colecionar os valores de decisão para um único processo. Este procedimento é apresentado na seção 6.1.2.

O ADC implementa dois tipos de terminação de protocolo: a terminação assíncrona e a terminação síncrona. A terminação de protocolo é implementada através da determinação de um limite máximo no tempo de entrega das mensagens. Tal

procedimento, geralmente é implementado para terminação síncrona de protocolo. Conforme mencionado na seção 2, o ADC além de introduzir limitação no tempo de entrega das mensagens para modelos síncronos, também introduz um limite máximo de tempo, dentro do qual as mensagens devem ser entregues para modelos assíncronos. Esta limitação é dada por intermédio de um *timeout*. O ADC considera um *timeout* razoavelmente grande, o que não compromete a característica assíncrona do modelo.

Quanto à ordenação de mensagens, o ADC considera quatro tipos, conforme apresentado em no item 2.1.1, são eles: sem ordenação, ordenação FIFO, ordenação total e ordenação causal. Aos modelos cujo tipo de ordenação de mensagens escolhido é sem ordenação ou FIFO, não há um algoritmo específico, mesmo porque tais tipos de ordenação não exigem algoritmos. A ordenação total é implementada através do *timestamp*, conforme explicado na seção 2.1.1.3. O ADC considera um *timestamp*, que por definição é um número global, cuja seqüência fornece a ordenação total.

Ordenação causal, no ADC, é conseguida através da aplicação dos conceitos relativos à causalidade potencial [LAM 78] [COU 94], descritos na seção 3.3.5. A relação de causalidade potencial também é conhecida na literatura como relação aconteceu-antes ("*happened-before*"). Para a implementação deste tipo de ordenação o ADC, além de utilizar um número de seqüência para as mensagens, adiciona um *label* (alfanumérico), o qual introduz a relação exigida. Por exemplo, todas as mensagens cujo *label* é B devem ser ordenadas segundo o *timestamp*. Este *label*, juntamente *timestamp*, é adicionado às mensagens estabelecendo, com isso, a ordenação causal entre mensagens que apresentam o mesmo *label*.

Conforme mencionado anteriormente, a execução do ADC é composta basicamente por duas etapas: experimentação e análise. No capítulo anterior foi apresentada a descrição dos módulos do ambiente, onde também foram abordados aspectos de implementação. Neste capítulo será apresentada a descrição detalhada da execução das etapas de experimentação e análise.

Após descritas as etapas que compõem a execução do ADC, é exibida a simulação de um modelo proposto, ou seja, serão mostradas as etapas de experimentação e análise para um determinado modelo.

6.1 Experimentação

A etapa de experimentação consiste na simulação da execução do modelo proposto. Para que seja possível a execução do ADC deve-se ativar a ferramenta na qual o mesmo foi desenvolvido, ou seja o sistema HetNOS. A inicialização de uma aplicação distribuída HetNOS, tal como o ADC, pode se dar de duas formas, conforme descrito no Anexo A-1 [BAA 95a]. Na primeira forma, as aplicações são invocadas a

partir da linha de comando de um *shell* do *Unix* (*csh*), enquanto na segunda forma, os comandos são chamados a partir do *shell* do HetNOS (*hsh*).

O modo pelo qual é inicializado o ADC, como aplicação distribuída HetNOS, é a partir do seguinte comando:

```
1 hsh_p0_polaris#pitthan: ~/HetNOS/prog % h med princ
```

De acordo com a linha de comando acima, *h* é o comando para a execução da aplicação distribuída HetNOS, no caso o ADC, cujo programa principal é denominado *princ* . Esta execução se dá através da criação local do processo denominado *med*, contração de Módulo Entrada de Dados, que é o módulo inicial do ambiente.

A execução do ADC está vinculada à especificação de um conjunto de informações definido como modelo proposto. O conjunto de informações correspondentes ao modelo proposto diz respeito ao número de processos que participarão da simulação do modelo, o número de estações nas quais o modelo irá executar, os tipos de terminação de protocolo e ordenação de mensagens a serem implementados e os tipos de falhas a serem suportados pelo modelo.

É de responsabilidade do Módulo Entrada de Dados a coleta dos dados referentes ao modelo proposto. Com base nestes dados, o ADC realiza uma verificação das características propostas. Esta verificação consiste na avaliação do modelo mediante às necessidades e restrições de implementação impostas pelo ADC. Tal fato se deve porque o ADC, apesar de ser caracterizado como um ambiente genérico para experimentação e avaliação de protocolos de difusão confiável, pode não ser capaz de executar determinadas classes de modelos.

Estando o modelo proposto consistente com a especificação de entrada aceita pelo ADC, o Módulo Entrada de Dados, módulo inicial do sistema, requisita ao HetNOS a criação do segundo módulo do ADC, o Módulo Protocolo.

Ao ser criado pelo Módulo Entrada de Dados, o Módulo Protocolo recebe como parâmetro as informações referentes ao modelo proposto. A primeira tarefa a ser realizada pelo Módulo Protocolo é a composição da rede.

A realização da composição da rede implica na criação dos processos que participarão da simulação do modelo. Esta criação compreende a determinação de dois aspectos: a determinação do local onde os processos irão rodar e a determinação de qual rotina tais processos irão executar. O primeiro aspecto a ser determinado, o qual envolve a criação e execução dos processos, deve considerar o balanceamento de carga.

A tarefa de balanceamento de carga consiste na distribuição dos processos ao longo da rede. Tal tarefa pode ser realizada tanto a nível de usuário como a nível de

sistema operacional. Quando realizada a nível de sistema operacional, a distribuição da carga fica a encargo do HetNOS, que utiliza-se de seus recursos para a determinação local de onde os processos irão executar. Já o balanceamento de carga realizado a nível de usuário deve levar em consideração as máquinas nas quais o HetNOS está atualmente ativo. A determinação destas máquinas corresponde a uma etapa realizada antes da simulação do modelo: a etapa de ativação do sistema HetNOS. No Anexo A-1 encontram-se descritos os passos necessários para a etapa de ativação do HetNOS.

Outro aspecto a ser determinado refere-se às rotinas as quais os processos irão executar. O ADC indica, para cada processo criado, qual rotina deve ser chamada.

O ADC é implementado sob o conceito de centralização, sendo assim, para cada conjunto de processos do modelo proposto, deve haver um processo centralizador. Este processo, denominado processo coordenador, é o responsável pelo gerenciamento da simulação da execução de toda a experimentação.

A eleição do coordenador é feita aleatoriamente a cada simulação do modelo. Um coordenador é eleito para realizar suas funções durante toda a experimentação, a menos que o mesmo falhe. Caso isso aconteça, é eleito um novo coordenador, conforme descrição posterior.

O processo eleito como coordenador executa uma rotina diferente dos demais processos. Isto devido às atribuições de um e outro serem diferentes. A rotina a ser executada pelo processo coordenador dá origem a um outro módulo do ADC, o Módulo Externo. Os demais processos executam rotinas do próprio Módulo Protocolo. O conjunto destes processos forma o que se chama de *grupo de difusão*. A figura 5.5 ilustra o grupo de difusão, bem como o processo coordenador.

O Módulo Protocolo cria uma lista de processos que corresponde à fusão do grupo de difusão ao processo coordenador. Esta lista, denominada *proc_list*, é passada como parâmetro para os demais módulos no momento da criação dos mesmos, juntamente com as informações referentes ao modelo proposto.

Durante a experimentação do modelo distingue-se duas fases: a difusão de mensagens e a determinação do consenso. Para cada mensagem difundida deve haver um procedimento de concordância sobre a mesma. Aspectos de implementação destas duas fases são descritas a seguir.

6.1.1 Implementação - Fase de Difusão

De acordo com o tipo de ordenação de mensagens exigido pelo modelo, o ADC seleciona o tipo de difusão de mensagens que melhor se adapta. Por definição, o ADC implementa a difusão de mensagens tendo em vista a propriedade de confiabilidade

(seção 3.2). Sendo assim, independente do tipo de ordenação de mensagens a ser implementado, a característica de entrega atômica de mensagens deve ser preservada.

Em difusão de mensagens, confiabilidade é alcançada com o conceito de entrega atômica de mensagem, cuja premissa é que todos os processos recebam a mensagem difundida, ou nenhum deles a aceite.

6.1.1.1 Difusão não-ordenada e Difusão FIFO

Aos modelos sem obrigação de ordenação ou com ordenação FIFO, o ADC implementa difusão confiável, conforme a classificação descrita em 3.3.3. Neste tipo de difusão, a única exigência é a confiabilidade na entrega das mensagens. Na implementação da difusão confiável pelo ADC, todos os processos estão aptos a difundir suas mensagens para os demais. A única restrição imposta é que a difusão de mensagens entre o processo emissor e o processo coordenador deve ser confiável. Isto é devido ao coordenador ser responsável pela bufferização de todas as mensagens difundidas pela rede. Este procedimento é necessário para assegurar a propriedade de confiabilidade.

6.1.1.2 Difusão Total

Para modelos cuja ordenação de mensagens exigida corresponde a ordenação total, o ADC implementa difusão atômica de mensagens. Neste tipo de difusão, conforme descrito em 3.3.4, além de ser exigida a propriedade de confiabilidade na entrega das mensagens, há também a exigência de ordenação total das mesmas. O ADC implementa ordenaçào total de mensagens através da introdução de um número de seqüência global nas mesmas. Este número de seqüência, denominado *timestamp*, é introduzido pelo processo coordenador, que mantém o controle das mensagens difundidas.

Em modelos nos quais difusão atômica é aplicada, a fase de difusão inicia no momento em que processos pertencentes ao grupo de difusão requisitam ao coordenador permissão para difusão. O coordenador seleciona um dos pedidos e envia uma mensagem ao requisitante contendo a permissão para difusão. A permissão para difusão é composta pelo número de seqüência (*timestamp*) que deve ser anexado à mensagem a ser difundida.

Ao receber a mensagem cujo conteúdo é um *timestamp*, o processo requisitante reconhece que conseguiu permissão para difusão. Desta forma, o processo requisitante envia a mensagem com o *timestamp* exigido para todos os processos da *proc_list*, a qual inclui o coordenador.

Cada processo do grupo de difusão, ao receber a mensagem, executa o algoritmo de ordenação de mensagens de acordo com a seqüência do *timestamp*. O processo coordenador, ao receber a mensagem, armazena-a no *buffer*. Este procedimento tem como objetivo o atendimento a futuros pedidos de retransmissão de mensagens. Como o ADC estabelece que a difusão de mensagens entre processos emissores e o processo coordenador deve ser confiável, é certo que todas as mensagens armazenadas no *buffer* estão corretas.

6.1.1.3 Difusão Causal

Aos modelos cuja exigência de ordenação corresponde a ordenação causal o ADC implementa difusão causal, conforme descrito em 3.3.5. Neste tipo de difusão são exigidas as propriedades de confiabilidade e causalidade potencial no envio das mensagens. De acordo com a seção 3.3.5, o ADC implementa difusão causal utilizando os conceitos de causalidade potencial introduzidos por [LAM 78]. Segundo [LAM 78], somente os eventos que estabelecem alguma relação de causa e efeito devem ser ordenados.

Conforme mencionado anteriormente, o ADC implementa ordenação causal com a introdução de um *label* às mensagens que devem estabelecer uma relação de causalidade potencial. Desta forma, os processos somente requisitam permissão para difusão de mensagens se for necessária a ordenação causal das mesmas. Neste caso o processo deve informar o *label* da mensagem a ser difundida. O processo coordenador mantém controle sobre todo o conjunto de mensagens difundido. Sendo assim, o coordenador envia ao processo requisitante o número de seqüência (*timestamp*) que deve ser anexado à mensagem.

O processo requisitante, ao receber o *timestamp* para o *label* solicitado, anexa-os à mensagem e a difunde para os demais processos. De forma análoga às difusões descritas anteriormente, o ADC implementa a difusão entre processos emissores e o processo coordenador de forma confiável. Isto porque o coordenador é o responsável pelo armazenamento da mensagem.

Os processos do grupo de difusão detectam a perda de alguma mensagem de difusão de duas formas: através do seqüenciamento de mensagens (*timestamp*), no caso de difusão atômica ou causal, ou através da limitação no tempo de entrega das mensagens. Esta limitação está geralmente associada a modelos com característica síncrona. Entretanto, conforme mencionado anteriormente, o ADC introduz um *timeout* como limite máximo no tempo de entrega de mensagens para modelos assíncronos.

Detectada a perda de alguma mensagem, os processos do grupo de difusão solicitam ao processo coordenador a retransmissão da mesma. O processo coordenador,

ao receber uma mensagem com pedido de retransmissão, realiza uma pesquisa no *buffer* e envia a mensagem para o processo requisitante.

O procedimento de liberação de mensagens do *buffer* é realizado periodicamente, com base em um tempo pré-fixado. Este procedimento consiste na verificação, junto aos processos do grupo de difusão, de quais mensagens foram recebidas por todos os processos. Mensagens estas cujo armazenamento somente é necessário enquanto todos os processos do grupo de difusão não acusarem recebimento.

6.1.2 Implementação - Fase de Consenso

Todas as mensagens cujo recebimento foi acusado por todos os processos da *proc_list* devem então passar pela fase consenso. Nesta fase é estabelecida a concordância sobre o valor difundido para cada mensagem. O Módulo Protocolo, através do Sub-Módulo Consenso, executa o procedimento de determinação do consenso. Sendo assim, o Sub-Módulo Consenso requisita ao processo coordenador o valor decidido para cada mensagem. Com este valor, o Sub-Módulo Consenso difunde uma mensagem para os processos do grupo de difusão requisitando uma resposta sobre o valor decidido para a mensagem de difusão. Esta resposta consiste em um ACK, caso o processo concorde com o valor decidido, ou um NACK, caso contrário. Com base nos ACK's e NACK's recebidos o Sub-Módulo Consenso realiza uma votação pela maioria. Desta forma, o Sub-Módulo Consenso obtém com um valor de decisão para a mensagem. Este algoritmo de consenso funciona perfeitamente caso o processo coordenador apresente o valor correto para todas as mensagens difundidas. Caso contrário, tanto o algoritmo de consenso como o procedimento de retransmissão de mensagens podem levar a inconsistências.

Falhas dos processos do grupo de difusão, provocadas pelo procedimento de introdução de falhas, são manipuladas pelo processo coordenador. Este detecta a falha de algum processo durante a verificação periódica de quais mensagens foram recebidas por todos os processos do grupo de difusão. Ao detectar uma falha, o processo coordenador deve reorganizar a lista de processos ativos (*proc_list*), difundindo-a para todos os processos envolvidos na experimentação do modelo.

Por outro lado, falhas introduzidas no processo coordenador podem ser detectadas pelos processos do grupo de difusão, devido à ausência de respostas tanto a pedidos de permissão para difusão como a pedidos de retransmissão de mensagens. Outro ponto de detecção de falhas do processo coordenador corresponde ao Sub-Módulo Consenso. Este pode detectar a falha do processo coordenador ao requisitar o valor decidido para uma mensagem. Neste caso, o Sub-Módulo Consenso deve acionar o Módulo Protocolo para que este realize a eleição do novo coordenador. Se, por outro lado, o Sub-Módulo Consenso detectar a falha de algum processo do grupo de difusão,

cabe a ele a tarefa de avisar ao processo coordenador para que este providencie a reorganização da *proc_list*.

A eleição de um novo coordenador é realizada com base no conjunto de mensagens apresentado por cada processo. Ao processo com o conjunto de mensagens mais bem definido é atribuída a responsabilidade de ser o novo processo coordenador. É responsabilidade do novo processo coordenador a coleta do último *timestamp* recebido por cada processo nos casos de difusão atômica ou causal. Em difusão não-ordenada ou difusão FIFO, o novo coordenador corresponde ao processo que apresentar o maior número de mensagens. É de responsabilidade do novo coordenador o estabelecimento de um ponto de sincronismo. Desta forma, o novo coordenador detecta quais as mensagens que foram recebidas pelos processos do grupo de difusão. O novo coordenador deve, então, divulgar estas mensagens aos respectivos processos, até que todos apresentem o mesma visão sobre o conjunto de mensagens difundidas. A partir deste ponto, o novo coordenador está apto ao reinício da fase de difusão, reiniciando o procedimento normal da experimentação.

No decorrer da experimentação, tanto as mensagens de envio como as mensagens de recepção devem receber tratamento. Este tratamento é oferecido pelo Sub-Módulo Trata-Mensagens. Isto porque as mensagens no ADC obedecem um padrão específico: (*origem*, *timestamp*, *nro_msg*, *tipo_msg*, *conteúdo*), o qual introduz a necessidade de uma tabela de códigos. Estes códigos identificam o tipo da mensagem. De acordo com o tipo de mensagem poda haver um ou mais parâmetros, os quais são apresentados no campo conteúdo.

O procedimento de geração das falhas exigidas pelo modelo proposto é realizado pelo Módulo Introdução de Falhas. Tal procedimento é executado através de mensagens. O Módulo Introdução de Falhas utiliza-se de uma tabela com os códigos de falha. Esta mesma tabela serve de base para o Sub-Módulo Trata-Falhas realizar o tratamento e recuperação das falhas inseridas pelo Módulo Introdução de Falhas. O tratamento e recuperação das falhas introduzidas pelo ADC, as quais seguem a classificação proposta por [CRI 91], são manipuladas pelo Sub-Módulo Trata-Falhas. Por definição, um processo que sofre falha de *crash* não se recupera. A nível de simulação, um processo que sofre falha da *crash* deve ser eliminado do sistema.

A recuperação de um processo que sofre tanto falha por omissão como falha de temporização segue a mesma filosofia. Isto porque as falhas de temporização, por definição do ADC, somente são implementadas por atraso, fazendo com que estas se assemelhem a falha por omissão.

No ADC, uma falha bizantina somente é descoberta quando da determinação do consenso. Portanto, fica a encargo do Sub-Módulo Consenso a retificação dos valores dos processos faltosos no caso de impossibilidade de consenso.

6.2 Análise

Conforme mencionado anteriormente, o ADC é executado basicamente em duas etapas: experimentação e análise. A etapa de experimentação, descrita na seção anterior, compreende a simulação da execução do modelo proposto. A partir de um conjunto de entradas, o definido com modelo proposto, é realizada a simulação de um protocolo de difusão confiável. Da etapa de experimentação são extraídos dados, os quais são utilizados como base para a realização de uma análise do modelo proposto.

A figura 6.1 mostra o fluxo de execução do ADC. Com base em um modelo proposto, cujas informações estão contidas na tabela 6.1, é realizada a etapa de experimentação. Como produto da experimentação são obtidos dados relativos à simulação do modelo. Estes dados, cuja especificação encontra-se na tabela 6.2, são o objeto da análise do ADC. Esta análise é o resultado fornecido pelo ADC para o modelo proposto.

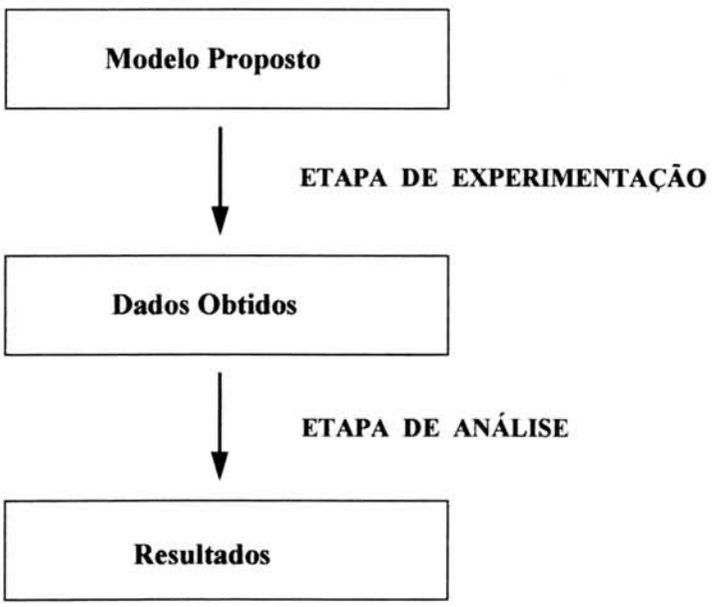


FIGURA 6.1 - Fluxo de Execução do ADC

Com base nos dados obtidos na primeira etapa, a experimentação, o ADC faz uma análise, de acordo com parâmetros pré-estabelecidos, sobre o comportamento do modelo frente às exigências de alta confiabilidade, baixa complexidade, baixo custo e alto desempenho. Tais exigências são medidas em termos de tráfego de mensagens, *bufferização* e ordenação de mensagens, suporte a falhas e determinação de consenso.

A implementação da etapa de análise do modelo proposto é realizada com base no estudo apresentado em [BAR 94], o qual envolve os protocolos propostos por [CHA 84], [BIR 87], [BAB 87] e [CRI 85], assim como idéias de [GAR 82], [FIS 85], [KAA

89], [BIR 91], [COU 94], [JAL 94], [SIN 94]. Tais idéias também se relacionam à protocolos de difusão confiável. Este embasamento possibilitou a implementação do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*.

TABELA 6.1 - Informações Referentes ao Modelo Proposto

número de processos que participarão da simulação do modelo
número de estações nas quais os processo irão executar
tipo de terminação de protocolo a ser implementado
tipo de ordenação de mensagens a ser implementado
tipos de falhas as quais o modelo deve suportar

Em se tratando de um ambiente genérico para experimentação e análise de protocolos de difusão confiável, tanto os dados obtidos como resultado da experimentação, como os parâmetros utilizados para análise, são determinísticos. Desta forma independentemente do modelo proposto, serão executados os mesmos métodos nas etapas de experimentação e análise.

A seguir é apresentada a descrição do significado dos dados obtidos com a experimentação do modelo, os quais encontram-se listados na tabela 6.2.

TABELA 6.2 - Relação dos Dados Obtidos com a Experimentação do Modelo

número de mensagens de difusão trocadas entre os processos
número de mensagens emitidas para a determinação do consenso
número de mensagens exigidas para a implementação da ordenação
número de mensagens de difusão perdidas ou não entregues ao destino
número de mensagens de controle (ACK's e NACK's) enviadas
número de mensagens de controle perdidas ou não entregues ao destino
número de mensagens de retransmissão enviadas
número de mensagens de falha enviadas
número de mensagens armazenadas em *buffer*
número de mensagens emitidas durante o processamento do modelo

- *número de mensagens de difusão trocadas entre os processos*: este número indica o número de mensagens necessárias para que todos os processos recebam a mensagem difundida. A estimativa para este número varia de acordo com o tipo de difusão de mensagens implementado para o modelo. Esta determinação está vinculada ao tipo de ordenação de mensagens exigido. Modelos que implementam ordenação causal de mensagens apresentam um menor número de mensagens trocadas a cada difusão do que modelos que implementam ordenação total. Entretanto, nos modelos aos

quais nenhuma obrigação de ordenação é exigida, este número apresenta-se bem mais reduzido. Em qualquer tipo de difusão, as mensagens de retransmissão representam um aumento neste número.

- *número de mensagens emitidas para a determinação do consenso*: este número determina o número de mensagens que devem ser difundidas para que todos os processos cheguem a uma concordância sobre o valor difundido. Este número é dependente do número de processos falhos existentes quando da realização do consenso.

- *número de mensagens exigidas para a implementação da ordenação*: este número indica o número de mensagens necessárias para que todos os processos executem o algoritmo de ordenação de mensagens exigido pelo modelo. Este número corresponde ao número de mensagens trocadas durante a fase de difusão, descrito anteriormente.

- *número de mensagens de difusão perdidas ou não entregues ao destino*: indica o número de mensagens correspondentes à fase de difusão que não conseguiram alcançar corretamente o destino. Tal fato se deve a ocorrência de falhas. Portanto, a determinação deste número está associada ao número de falhas introduzidas no modelo.

- *número de mensagens de controle (ACK's e NACK's) enviadas*: este número corresponde ao número de confirmações de valor enviados pelos processos durante a fase de consenso. Devido ao ADC trabalhar com o esquema de reconhecimento negativo ("*negative acknowledgement*"), conforme mencionado anteriormente, na fase de difusão de mensagens não há emissão de mensagens de controle. Entretanto, o algoritmo de consenso utiliza-se deste recurso para a realização da votação pela maioria, isto é, para a determinação da concordância entre os membros do sistema livres de falha.

- *número de mensagens de controle perdidas ou não entregues ao destino*: indica o número de mensagens de controle que não conseguiram alcançar corretamente o destino, provavelmente devido à ocorrência de falhas. Assim como o número de mensagens de difusão perdidas, para este número não há uma estimativa, uma vez que também depende do número de falhas introduzidas no modelo.

- *número de mensagens de retransmissão enviadas*: este número corresponde ao número de mensagens que o processo coordenador teve que retransmitir aos processos do grupo de difusão, devido à ocorrência de falhas nos mesmos.

- *número de mensagens de falha enviadas*: este número corresponde ao número de falhas introduzidas no modelo, por meio de mensagens, pelo Módulo Introdução de Falhas. Este número é gerado pelo ADC de maneira aleatória.

- *número de mensagens armazenadas em buffer*: este número é dependente tanto do tempo fixado para a verificação periódica das mensagens recebidas, a qual é realizada pelo coordenador, como do fluxo de mensagens enviado. Quanto maior o intervalo de tempo que o coordenador espera para fazer a verificação periódica, maior o número de mensagens que devem ficar armazenadas no *buffer*. De maneira análoga, quanto mais contínuo o fluxo de mensagens enviado, maior deve ser o tamanho do *buffer*.

- *número de mensagens emitidas durante o processamento do modelo*: este número envolve as mensagens de difusão, as mensagens de falha e as mensagens emitidas para a determinação do consenso.

O ADC pode exibir os números acima descritos de duas formas: número médio e número total. O número médio está relacionado particularmente a uma fase de difusão ou consenso, ou seja, a difusão ou consenso de uma determinada mensagem. Já o número total abrange toda a fase de difusão ou consenso, que envolve o conjunto de mensagens difundido para o modelo.

Os dados extraídos servem como base para a etapa de análise. Esta etapa é realizada de acordo com os parâmetros descritos na tabela 6.3. Esta análise será realizada de acordo com as exigências à respeito de confiabilidade, complexidade, custo e desempenho. A especificação dos parâmetros listados na tabela 6.3 é apresentada nos itens a seguir.

TABELA 6.3 - Relação dos Parâmetros para Análise do Modelo

bufferização de mensagens
ordenação de mensagens
determinação do consenso
tráfego de mensagens

6.2.1 *Bufferização* de Mensagens

O ADC implementa a *bufferização* de mensagens com o objetivo de armazenar as mensagens de difusão até que todos os processos as tenham recebido. A *bufferização* de mensagens visa assegurar que eventuais pedidos de retransmissão de mensagens sejam devidamente atendidos.

O controle de *bufferização* de mensagens é de responsabilidade do processo coordenador. Cada mensagem de difusão por ele recebida deve ser armazenada no *buffer*. Periodicamente, o processo coordenador deve verificar, junto ao grupo de difusão, quais as mensagens cujo recebimento foi efetuado por todos os processos do grupo de difusão. Estas podem, portanto, ser descartadas do *buffer*. Isto porque, para

estas mensagens não chegarão pedidos de retransmissão, uma vez que todos os processos já as receberam.

A implementação de *bufferização* de mensagens pelo ADC visa introduzir um maior grau de confiabilidade na experimentação do modelo. Contudo, o benefício não vem sem o custo. A introdução de um *buffer*, por mais limitado que seja, implica em um custo adicional de recurso e gerência. Tarefas como: armazenamento de mensagens, recuperação de mensagens, verificação periódica do conjunto de mensagens recebido pelos demais processos e remoção de mensagens exigem custo em termos de tempo, recursos e gerenciamento. Custo em termos de recursos e gerenciamento, por sua vez implicam em aumento de complexidade no modelo.

Quando o custo é medido em termos de tempo introduz-se queda de desempenho, uma vez que o desempenho em protocolos de difusão confiável é medido com relação ao tempo dispendido com a execução dos mesmos. Já o custo no que se refere ao gerenciamento implica em aumento de complexidade. Isto porque deve haver todo um procedimento de manipulação de *buffer*, que envolve as tarefas acima mencionadas. Outro aspecto a ser considerado diz respeito aos recursos exigidos com a utilização da *bufferização* de mensagens. Tais recursos envolvem alocação de espaço de memória, introduzindo, com isso, custo e complexidade.

O ADC avalia a necessidade de *bufferização* de mensagens com base no número de mensagens de retransmissão enviadas quando da simulação do modelo proposto. Este número indica quantas vezes foi realmente necessária a utilização de um *buffer*.

Simulações de modelos que não ocasionam pedidos de retransmissão de mensagens poderiam apresentar diminuição tanto no custo como na complexidade e aumento no desempenho. Tal suposição considera a não utilização de *bufferização* de mensagens. Contudo, esta constatação não é determinante. Para um mesmo modelo pode não haver necessidade de *bufferização* em uma execução específica, porém simulações de execução exaustivas deste mesmo modelo podem comprovar esta necessidade.

A simulação de um modelo pode ser aproveitada para a determinação do tamanho do *buffer*. Este tamanho deve ser medido de acordo com o número médio de mensagens armazenadas. Este procedimento é necessário para evitar desperdícios na tarefa de alocação de espaço de memória.

Embora o número médio de mensagens armazenadas em *buffer* seja dependente do período no qual estas ficam armazenadas, pode-se fazer uma projeção utilizando um período de tempo fixo. Tal período de tempo indica o intervalo dentro do qual o processo coordenador deve fazer uma verificação das mensagens já recebidas por todos os processos do grupo de difusão. Cabe ressaltar que este período de tempo deve ser inversamente proporcional ao número de mensagens difundidas de forma contínua. Isto

se explica pelo fato de que quanto maior o número de mensagens continuamente difundidas, maior deve ser o número de mensagens armazenadas em *buffer*. Como o *buffer* possui capacidade limitada, o fluxo armazenamento-remoção deve ser constante, ou o tamanho do *buffer* deve ser consideravelmente grande.

6.2.2 Ordenação de Mensagens

O ADC implementa quatro tipos de ordenação de mensagens. Compreendem a implementação destes tipos de ordenação de mensagens os algoritmos sem ordenação, algoritmos com ordenação FIFO, algoritmos com ordenação total e algoritmos com ordenação causal, descritos na seção 2.1.1. A ordenação de mensagens tem como objetivo garantir que todos os processos recebam as mensagens numa mesma ordem. Esta ordem estabelece uma espécie de sincronização entre os processos. Desta forma, todos os processos terão a mesma visão sobre o conjunto de mensagens difundido, tornando, com isso, o sistema mais confiável e consistente.

Os algoritmos de ordenação de mensagens do ADC visam introduzir um maior grau de confiabilidade na experimentação do modelo, bem como atender às exigências de sincronização impostas pelo mesmo. Tais algoritmos apresentam níveis de complexidade diferentes. A escolha destes algoritmos está relacionada ao tipo de aplicação para o qual se destina.

Em modelos cujo envio ou recepção de mensagens não estabelece qualquer precedência, tanto algoritmos sem ordenação como algoritmos com ordenação FIFO podem ser utilizados. Isto se deve ao fato de que estes algoritmos apresentam menor grau de complexidade e, portanto, menor custo. Para estes modelos o ADC implementa difusão confiável, a qual exige somente a propriedade de confiabilidade.

Já em modelos nos quais o envio ou recepção de mensagens estabelece pelo menos alguma relação de causa e efeito, devem ser aplicados algoritmos de ordenação causal ou ordenação total. Estes algoritmos caracterizam-se por apresentar tanto confiabilidade como ordenação de mensagens ao modelo. Contudo, implicam em aumento de complexidade e custo. Algoritmos com alto grau de complexidade tendem a degradar o desempenho do modelo.

Um fato determinante na ordenação de mensagens diz respeito ao número de mensagens exigidas para a execução do algoritmo. Aplicações que utilizam-se de algoritmos mais complexos (por exemplo, ordenação total), nos quais pode-se reduzir a obrigação de ordenação, o número de mensagens trocadas entre os processos pode diminuir significativamente. Com isso, pode-se chegar a diminuição na complexidade e no custo, obtendo, desta forma, um ganho no desempenho, sem que a confiabilidade seja alterada.

6.2.3 Determinação do Consenso

A deteminação de uma concordância entre os membros do sistema livres de falhas é uma característica desejável em técnicas de difusão confiável. Mesmo porque estas técnicas, quando aplicadas à sistemas distribuídos, apresentam alta confiabilidade como exigência. Para tanto, faz-se necessária a implementação de um algoritmo que possibilite estabelecer esta concordância. O algoritmo de consenso não deve inviabilizar a utilização do modelo com relação ao número de mensagens exigidas para a determinação da concordância.

O ADC implementa um algoritmo de consenso classificado como centralizado. Isto se deve ao fato de que a tarefa de colecionar valores de decisão entre todos os processos é atribuída a um único processo. Embora esta técnica não seja ideal, em se tratando de sistemas distribuídos confiáveis, a necessidade de implementação de consenso introduziu a sua utilização.

A implementação do algoritmo de consenso impôs a restrição no que diz respeito ao envio de mensagens faltosas para o processo coordenador. Mensagens faltosas correspondem a mensagens de difusão que sofreram alteração devido a uma falha bizantina. Cabe ressaltar que esta restrição também foi imposta pelo procedimento de retransmissão de mensagens, cuja execução tornaria o sistema inconsistente.

A forma com que o ADC avalia o algoritmo de consenso é através do número de mensagens necessárias para a determinação do mesmo. Este número é dependente do número de falhas introduzidas no sistema. De acordo com a imposição do algoritmo, estas falhas não devem atingir o processo coordenador, estando desta forma restritas à aplicação nos processos do grupo de difusão. Com isso, o ADC verifica se o modelo proposto, apesar da introdução de falhas, consegue alcançar um consenso sobre o conjunto de mensagens difundido.

6.2.4 Tráfego de Mensagens

Em um modelo de difusão confiável é de extrema importância a determinação do desempenho do mesmo. Tal determinação depende fortemente do número de mensagens envolvidas durante a simulação do modelo. Um modelo para difusão confiável não deve inviabilizar sua utilização devido ao excessivo número de mensagens por ele exigido. Um exemplo de implementação inviável devido ao expressivo número de mensagens envolvido é o Problema dos Generais Bizantinos, descrito na seção 3.4.1. O algoritmo proposto por [LAM 82], apesar de ser ótimo para os propósitos do problema, torna-se inviável na prática pelo número de mensagens de difusão exigidas.

Neste contexto, o ADC visa analisar a viabilidade de determinados modelos de acordo com o número total de mensagens por eles exigidas. Desta forma, pode-se estabelecer parâmetros mínimo e máximo, dentro dos quais o número de mensagens envolvido na simulação do modelo deve se adequar.

Os trabalhos desenvolvidos em [BAR 94] e [BAR 95] serviram de base para a determinação de parâmetros aceitáveis em difusão confiável. Por parâmetros aceitáveis subentende-se parâmetros dentro dos quais um modelo para difusão confiável não inviabilize a sua utilização no que se refere à troca de mensagens.

O tráfego de mensagens em difusão confiável envolve não somente a troca de mensagens de difusão, como também as mensagens associadas à ordenação destas mensagens, e até mesmo as mensagens associadas à determinação do consenso. A determinação de limites razoáveis para estes números de mensagens deve obedecer um critério de análise, o qual deve ser atribuído diferentemente a cada modelo, de acordo com às necessidades e restrições impostas pelo mesmo.

6.3 Simulação da Execução de um Modelo Proposto

A descrição do modelo proposto apresentada a seguir consiste na demonstração da utilização do ADC. Os dados obtidos com a experimentação deste modelo, bem como os resultados provenientes da análise do mesmo consideram a simulação da execução de um mesmo modelo. Entretanto, por esta experimentação ter sido realizada em uma rede na qual não se tem controle sobre a carga dos nodos, os dados apresentados consideram a média dos valores obtidos, após sucessivas simulações de execução de um mesmo modelo proposto.

O modelo proposto para a experimentação e análise consiste de 12 processos, os quais devem ser executados em três máquinas distintas. O tipo de terminação de protocolo a ser implementado pelo ADC deve ser a terminação assíncrona. O tipo de ordenação de mensagens deve adotar o algoritmo para ordenação total de mensagens. A semântica de falhas exigida para a simulação do modelo são as falhas por omissão.

O primeiro passo a ser executado corresponde à ativação do sistema HetNOS. De acordo com o Anexo A-1 os procedimentos necessários para a ativação do sistema HetNOS são apresentados no Anexo A-1.

A entrada de dados do ADC é ilustrada na figura 6.2. Após a entrada de dados é realizada a verificação dos mesmos. Esta verificação, conforme mencionado anteriormente, visa a confirmação de que o modelo proposto apresenta as características exigidas pelo ambiente.

Entrada de dados : Modelo Proposto

Número de Processos: **12**

Rede disponível: polaris
rigel
auriga
orion
vega

Usar todas as estações ou escolher (T/E): **E**

Que estações devem participar do processamento ? **polaris auriga vega**

Tipo de Ordenação de Mensagens: **T**
S: Sem ordenação
F: FIFO
C: Causal
T: Total

Tipo de Terminação de Protocolo: **A**
A: Assíncrona
S: Síncrona

Tipos de Falha a serem suportadas: **O** (para modelos assíncronos)
C: Crash
O: Omissão

FIGURA 6.2 - Entrada de Dados do ADC

Caso o modelo proposto esteja de acordo com a especificação exigida pelo ADC, este exibe uma mensagem vinda do HetNOS informando a criação do primeiro processo, ou seja, o processo pai (Módulo Entrada de Dados) do modelo.

Nesta etapa do processamento são exibidas tanto as mensagens do HetNOS, referentes ao controle do sistema sobre a aplicação distribuída, como as mensagens do próprio ADC. Por questões de simplicidade, esta parte do processamento foi abstraída da demonstração.

Dados Obtidos com a Experimentação do Modelo Proposto

Tipo de Difusão Implementado: Difusão Total
Número de mensagens difundidas: 8

- número de mensagens de difusão trocadas entre os processos:
  número médio: 19.25
  número total: 154

- número de mensagens emitidas para a determinação do consenso:
  número médio: 20.25
  número total: 162

- número de mensagens exigidas para a implementação da ordenação:
  número total: 154

- número de mensagens de difusão perdidas ou não entregues ao destino:
  número médio: 3.125
  número total: 25

- número de mensagens de controle (ACK's e NACK's) enviadas:
  número médio: 8.5
  número total: 68

- número de mensagens de retransmissão enviadas:
  número médio: 3.125
  número total: 25

- número de mensagens de falha enviadas:
  número médio: 6.625
  número total: 53

- número de mensagens armazenadas em buffer:
  número médio: 3.625
  número total: 6.125

- número de mensagens emitidas durante o processamento do modelo:
  número médio: 46.125
  número total: 369

FIGURA 6.3 - Dados Obtidos com a Experimentação do Modelo

Finalizada a etapa de experimentação do modelo são exibidos os dados obtidos com a simulação do mesmo. Estes dados encontram-se dispostos na figura 6.3. Os dados obtidos representam a média dos extraídos. Isto porque, para a obtenção destes valores foram realizadas várias simulações do modelo proposto. A partir dos dados obtidos com a experimentação do modelo, exibidos na figura 6.3, o ADC apresenta a análise efetuada sobre os mesmos. Os resultados obtidos com a análise dos dados da figura 6.3 são apresentados na figura 6.4.

Análise do Modelo Proposto - Resultados

Modelo Proposto: 12 processos
3 estações
Ordenação Total
Terminação Assíncrona
Falhas: Omissão

Tipo de Difusão: Difusão Total
Número de mensagens difundidas: 8

- número de mensagens de difusão trocadas entre os processos:
  número médio:
    ideal: 13
    máximo aceitável: 23
    MODELO: 19.25

  número total:
    ideal: 104
    máximo aceitável: 184
    MODELO: 154

- número de mensagens emitidas para a determinação do consenso:
  número médio:
    ideal: 24
    mínimo aceitável: 18
    MODELO: 20.25

  número total:
    ideal: 192
    mínimo aceitável: 148
    MODELO: 162

FIGURA 6.4 - Análise dos Dados Obtidos com a Experimentação do Modelo

- número de mensagens exigidas para a implementação da ordenação:
  - número médio:
    - ideal: 13
    - máximo aceitável: 23
    - MODELO: 19.25
  - número total:
    - ideal: 104
    - máximo aceitável: 184
    - MODELO: 154
- número de mensagens de controle (ACK's e NACK's) enviadas:
  - número médio:
    - ideal: 11
    - mínimo aceitável: 6
    - MODELO: 8.5
  - número total:
    - ideal: 88
    - mínimo aceitável: 48
    - MODELO: 68
- determinação do tamanho do buffer:
  - ideal: 5
  - mínimo aceitável: 4
  - ocupação MODELO: 3.625

FIGURA 6.4 - Análise dos Dados Obtidos com a Experimentação do Modelo

Os valores relativos à análise representam o número ideal esperado para o dado, os número mínimos e máximos aceitáveis, bem como o número apresentado durante as simulações do modelo.

7 Conclusão

Os sistemas de computação em desenvolvimento têm voltado sua atenção para a modelagem distribuída de sistemas. Com isso, as necessidades de confiabilidade tem aumentado, incentivando, assim, a pesquisa nesta área.

Esse trabalho enfatiza difusão confiável, uma técnica básica que possibilita alcançar um maior grau de confiabilidade em sistemas distribuídos. Difusão confiável é necessária sempre que se deseja disseminar, de forma confiável, uma mensagem para todos os processos de um sistema distribuído. Sua utilização se torna obrigatória sempre que processos precisam alcançar consenso e encontra larga aplicação, por exemplo, em diagnóstico de falhas em sistemas distribuídos e reconfiguração para isolamento ou acréscimo de nodos.

Vários protocolos de difusão confiável têm sido publicados na literatura. O objetivo desse trabalho foi estabelecer uma base de comparação entre esses protocolos, através de experimentação, executando os protocolos em um ambiente próximo de sua aplicação na prática. Com os protocolos sendo executados, podem ser obtidos parâmetros para análise que dependem fortemente de características dinâmicas do sistema distribuído e que, de outra forma, seriam impossíveis de ser obtidos por simples análise estática da descrição dos protocolos.

O suporte para implementação do ambiente é suprido pelo sistema operacional de rede heterogêneo HetNOS. Tal suporte é obtido através de suas primitivas para comunicação, gerenciamento e sincronização de processos. Os resultados alcançados através de análise dos protocolos no ambiente poderão vir a servir, por sua vez, para dotar o HetNOS de um maior grau de confiabilidade.

A adoção do sistema HetNOS como plataforma de desenvolvimento possibilitou vantagens no que se refere às primitivas por ele oferecidas, embora sua utilização tenha ocasionado algumas restrições na implementação do ADC. Tais restrições impuseram decisões as quais dificultaram a implementação de um ambiente cujo objetivo é a realização de uma análise. Estes fatores são devidos a deficiências no que diz respeito à depuração do sistema HetNOS. Um sistema não totalmente depurado pode tornar o projeto e implementação de aplicações bastante complexo.

Contudo, todo o desenvolvimento e implementação de um projeto trazem consigo algum aprendizado. Com relação ao ADC, ficou comprovado que a simulação de um ambiente para aplicações distribuídas poderia apresentar resultados mais satisfatórios. Isto se deve não somente pelas deficiências apresentadas pelo sistema HetNOS, mas também pela viabilidade de alcançar resultados mais precisos, uma vez que se abandonaria a idéia de implementação em uma rede de estações de trabalho. A implementação de um ambiente de análise sobre uma rede de estações conduz a

resultados algumas vezes totalmente discrepantes. Tal fato pode estar relacionado à carga dos nodos, diferentes velocidades de processamento das máquinas, etc, enfim, parâmetros sobre os quais não se pode ter um controle determinístico. A utilização de uma máquina exclusiva, no caso um PC, poderia conduzir a uma experimentação mais precisa, uma vez que fatores externos não teriam influência sobre a mesma.

Entretanto, conseguiu-se atingir os objetivos os quais foram propostos quando da etapa de projeto do *Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável (ADC)*. No que diz respeito ao *software* utilizado como plataforma de desenvolvimento, o HetNOS, a alternativa encontrada foi a implementação de um ambiente que estivesse de acordo com às restrições por ele impostas. Tal implementação deveria, contudo, preservar a característica analítica do ambiente. Outro fator a ser contornado se refere ao *hardware*, isto é, o recurso utilizado para a implementação. Conforme mencionado acima, uma implementação sobre um sistema de rede oferece desvantagens. Através da sucessão de testes e execução exaustiva dos modelos, os quais representam o objeto de análise do ADC, foi possível a obtenção de parâmetros, os quais por simples observação teórica não condiziriam com a realidade.

A partir de dados obtidos com a experimentação de um modelo proposto, etapa que recebe a denominação de experimentação, realiza-se a etapa de análise dos mesmos. Esta análise considera aspectos relacionados aos parâmetros de confiabilidade, complexidade, custo e desempenho proporcionados pelo modelo.

Do ponto de vista de confiabilidade, o ADC avalia basicamente três aspectos: o tipo de difusão de mensagens, que envolve o tipo de ordenação de mensagens implementado para o modelo, a *bufferização* de mensagens, a qual se relaciona à necessidade de retransmissão e a determinação do consenso, cujo número de mensagens envolvido não deve inviabilizar a utilização do modelo.

Quanto à difusão de mensagens, um modelo pode estar associado a três tipos: difusão atômica, difusão causal e difusão total. O tipo de difusão introduz um maior ou menor grau de confiabilidade ao modelo. Apesar da confiabilidade estar relacionada à propriedade de atomicidade, pode-se dizer que a ordenação de mensagens representa um fator determinante para o estabelecimento da mesma. Isto porque, algoritmos de ordenação de mensagens proporcionam uma visão única do conjunto de mensagens por todos os participantes da difusão.

A *bufferização* de mensagens corresponde a outro aspecto fundamental na determinação do grau de confiabilidade de um modelo. Isto se deve ao fato de que modelos que proporcionam um armazenamento temporário de mensagens oferecem tolerância a falhas sob o ponto de vista de redundância. O grau de confiabilidade proporcionado pela *bufferização* de mensagens está associado ao atendimento a pedidos de retransmissão de mensagens em caso de falhas.

O algoritmo de consenso visa oferecer confiabilidade a todos os processos envolvidos na difusão, sobre as mensagens difundidas. Desta forma, para cada mensagem deve haver uma concordância entre todos os processos livres de falha. A análise da execução de um algoritmo de consenso deve levar em consideração o número de mensagens trocadas para a determinação da concordância. Sendo assim, para este número deve-se estabelecer uma limitação, dentro da qual os processos devem chegar a alguma conclusão.

O segundo parâmetro sobre o qual o ADC avalia seus modelos se refere à complexidade. Um modelo deve apresentar uma complexidade aceitável para que sua utilização não venha a atingir outros parâmetros, tais como desempenho e custo. Desta forma o ADC avalia os mesmos aspectos avaliados sob o parâmetro de confiabilidade, ou seja: o tipo de ordenação de mensagens, que tem conseqüência direta no tipo de difusão das mesmas, a bufferização de mensagens e a determinação do consenso.

Complexidade, no contexto de ordenação de mensagens, implica na determinação das reais necessidades de ordenação do modelo em questão. Esta determinação tem influência direta no tipo de difusão de mensagens a ser implementado. Desta forma não se deve superestimar a necessidade de ordenação de cada modelo. Com isso, obtém-se um ganho com relação à complexidade do algoritmo de ordenação implementado. Além de tudo, a complexidade está vinculada diretamente ao custo do modelo.

Sob o aspecto de bufferização de mensagens o parâmetro de complexidade envolve tarefas relacionadas ao gerenciamento de recursos e mensagens. Esta tarefa, por sua vez, está relacionada tanto a determinação do tamanho do *buffer* como ao procedimento de armazenamento e liberação de mensagens.

O parâmetro de complexidade deve avaliar o comportamento do modelo com relação ao algoritmo de consenso. Isto porque, conforme mencionado anteriormente, o número de mensagens envolvidas com a determinação da concordância não deve inviabilizar a utilização do modelo como um todo.

No que se refere ao parâmetro de custo, o ADC visa avaliar quatro aspectos: o tipo de ordenação de mensagens, o procedimento de *bufferização* de mensagens, a determinação do consenso e o tráfego de mensagens envolvido.

Estes quatro aspectos podem ser avaliados sob o ponto de vista de apenas um: o tráfego de mensagens. Desta forma, o custo dos modelos é analisado com base no número de mensagens envolvidas na execução do mesmo. Isto inclui o número de mensagens envolvidas com o tipo de ordenação implementado, o fluxo de armazenamento, recuperação e liberação de mensagens do *buffer* e o número de mensagens necessárias para a obtenção do consenso. A que se considerar também o custo em termos de recursos de *hardware* exigido pelo *buffer*.

Quanto ao desempenho, o ADC analisa a degradação imposta pelo tráfego de mensagens quando da execução do modelo proposto. Conforme mencionado anteriormente, o tráfego de mensagens corresponde a todo o procedimento de experimentação do modelo, o qual envolve trocas de mensagens para a realização da ordenação de mensagens, trocas de mensagens relativas à *bufferização*, e as trocas de mensagens referentes à determinação do consenso.

A necessidade de um ambiente capaz de simular a execução de protocolos de difusão confiável para a realização de uma análise conduziu ao projeto do ADC. O ADC compreende o que se chama de aplicação distribuída HetNOS. O objetivo na implementação do ADC é viabilizar a sua utilização em sistemas distribuídos. Para tanto, deve-se avaliar os objetivos do sistema distribuído em questão, ou seja, deve-se avaliar o tipo de aplicação para a qual o sistema distribuído se destina. Com isso, tem-se idéia das exigências e restrições do sistema distribuído.

Anexo 1 Ativação do HET*NOS*

Esta seção tem como objetivo apresentar os requisitos básicos para a utilização do HetNOS como plataforma de desenvolvimento do ADC.

Como Instalar o HetNOS

Nesta seção serão descritos os procedimentos necessários para a instalação do HetNOS no sistema SunOS.

Para instalar o HetNOS em uma máquina, primeiramente deve ser criado um diretório exclusivo para o mesmo (denominado, preferencialmente HetNOS). A partir deste diretório, podem ser criados os diversos sub-diretórios do HetNOS, dependendo do propósito da instalação. No mínimo, devem ser criados o diretório dos executáveis, *bin*, e da documentação, *doc*. Para apenas utilizar o HetNOS não é necessário copiar os fontes do sistema. Naturalmente, uma vez criados os sub-diretórios, seus respectivos arquivos devem ser copiados para os mesmos [BAA 93].

É necessário criar uma variável de ambiente indicando em que diretório se encontram os executáveis do HetNOS. Tal variável de ambiente denomina-se `H_BIN`, e pode ser criada com o comando *Unix* `setenv`, tal como:

```
setenv H_BIN /tools2/HetNOS/bin
```

Este comando pode ser executado a partir do sinal de *prompt* do shell, mas aconselha-se que a linha acima seja incluída no arquivo *.cshrc*, o interpretador de comandos utilizado pelo sistema SunOS [BAA 93].

Outra alteração é incluir o diretório onde se encontram os arquivos executáveis do HetNOS na variável `PATH`. Uma vez definida a variável de ambiente `H_BIN`, o `PATH` pode ser modificado acrescentando-se o componente `$BIN` à linha que define o mesmo, tal como [BAA 93]:

```
set PATH  (/bin /usr/bin /usr/local/bin $H_BIN /usr/x11/bin)
```

Como Montar uma Rede HetNOS

O primeiro passo para montar uma rede HetNOS é escolher uma estação da rede para executar o Servidor de Boot (BS), por exemplo "polaris", e acionar pela linha de comando [BAA 95a]:

```
12 pitthan@polaris:~ % bootsv
```

O conteúdo impresso a partir da linha de comando acima apresenta as *features* da versão atual ou *release* do sistema, bem como uma mensagem indicando que o Servidor de Boot (BS) está pronto e como inserir estações na rede [BAA 95a].

A inicialização do Servidor de Boot é imediata. Logo a seguir, as máquinas podem ser adicionadas para compor a rede HetNOS. Para inserir uma máquina, basta executar o acionador local do sistema na estação em questão (comando "hetnos"), passando como parâmetro o nome da estação em que se encontra o Servidor de Boot (no caso acionando o HetNOS na mesma máquina em que se encontra o BS) [BAA 95a]:

```
15 pitthan@polaris:~ % hetnos polaris &
```

O conteúdo impresso ao acionar o comando acima é semelhante àquele impresso na *tty* do BS, enquanto que na tty do BS são apresentadas mensagens informando a nova configuração da rede HetNOS, que representa a inclusão de um nodo [BAA 95a].

A partir desse momento, aplicações HetNOS podem ser executadas na máquina "polaris". Para acrescentar novas máquinas à rede, o procedimento se repete (abrindo uma sessão em cada máquina remota com o comando *rlogin* e executando o iniciador local do sistema - "hetnos"). A rede está pronta quando surge a mensagem "HetNOS READY AT *nodo*" na *tty* onde foi ativada a nova instância do HetNOS [BAA 95a].

Como Abrir uma Sessão HetNOS

O HetNOS controla o acesso a seus serviços através de um esquema de sessões de usuário. Usuários devem ser cadastrados pelo super-usuário HetNOS antes de acessar o sistema [BAA 95a].

Um usuário pode abrir múltiplas sessões, alocando uma ou mais sessões em cada máquina. Existem dois modelos de trabalho em relação ao uso do HetNOS [BAA 95a]:

- um programa é executado a partir da linha de comando de um *shell* do *Unix*, então invoca a chamada *h_login( )*, utiliza serviços do HetNOS (incluindo criação de processos) e, ao terminar, executa *h_logout( )*, retornando ao *shell* do *Unix*;

- primeiramente invoca-se, a partir do *Unix*, um programa especial do HetNOS denominado *hlogin*, que requer a entrada do *username* e da senha. O *shell* do

HetNOS é executado e todas as aplicações de usuário são executadas a partir desse *shell*. Não há necessidade de um *login* a cada nova aplicação, bastando executar *h_init( )*.

Uma das diferenças entre um modelo e outro é que no primeiro aplicações são invocadas a partir da interface de comandos do *Unix* (o *csh*, por exemplo), enquanto que no outro os comandos são chamados a partir do *shell* do HetNOS, denominado *hsh* [BAA 95a].

De acordo com o primeiro modelo de trabalho, descrito anteriormente, para abrir uma sessão no HetNOS basta escrever uma aplicação que inicie com a chamada *h_login( )* e termine com *h_logout( )*. Todos os processos criados por esse programa devem normalmente executar *h_init( )* ao entrar e *h_exit( )* ao sair. Caso não seja chamado *h_logout( )* ao término do processo que executou *h_login( )*, o HetNOS detectará a saída da aplicação e assumirá que houve uma falha na mesma [BAA 95a].

Considerando o modelo que dispara aplicações a partir do *shell* do HetNOS, basta executar da linha e comando do *shell Unix* o programa *h_login( )*. Ele requisitará a entrada do username e da senha, verificará sua autenticidade e abrirá uma sessão de acordo com o tipo de usuário. A figura abaixo ilustra um exemplo de abertura de sessão [BAA 95a].

```
8 pitthan@polaris:~/HetNOS/prog %  hlogin pitthan
Password:

HetNOS - HetNOS Network Operating System

User pitthan is logged on HetNOS
Last logged in Fri Jan 30 13:02:40 1996 at auriga

hshell - The HetNOS Shell - Command Interpreter
? for help              v.1.5 - 4 July 1994
...now with aliases and history.

1 hsh_p0_polaris#pitthan:~/HetNOS/prog %
```

O *prompt* do *hsh* é composto pelas seguintes partes [BAA 95a]:

- número do comando no histórico de comandos;
- nome do processo que executa o *hsh* - nome este escolhido pelo porcesso de login, e formado pela cadeia "*hsh_*" seguida da identificação da *tty* e do nome da máquina;

- identificador de usuário dono do processo que executa o *hsh*;
- diretório corrente de trabalho;
- sinal de percento como separador.

Este *prompt* não é reconfigurável como acontece em interpretadores de comandos *Unix*.

Após a abertura de sessão o HetNOS está apto a atender solicitações referentes à execução de aplicações distribuídas. A ativação de uma aplicação distribuída, tal como o ADC, é realizada através do comando h, conforme mostrado a seguir:

```
1 hsh_p0_polaris#pitthan:~/HetNOS/prog % h med princ
```

De acordo com a linha de comando acima, *h* é o comando para a execução da aplicação distribuída HetNOS, no caso o ADC, cujo programa principal é denominado *princ*. Esta execução se dá através da criação local do processo denominado *med*, contração de Módulo Entrada de Dados, que é o módulo inicial do ambiente.

Bibliografia

[AMO 91] AMORIM, R. E. H. -B.; LOQUES, O. G. Protocolos de Difusão Confiável. *In:* SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 4., 1991, Gramado, RS. **Anais...** Porto Alegre: SBC, 1991. 272p.

[AND 93] ANDREWS, Gregory R.; OLSSON, Ronald A. **The SR Programming Language**: concurrency in practice. Redwood City: The Benjamin/Cummings, 1993. 344p.

[BAA 93] BARCELLOS, A. M. P. **O Sistema Operacional de Rede Heterogêneo HetNOS**. Porto Alegre: CPGCC da UFRGS, 1993. 300 p. Dissertação de Mestrado.

[BAA 93a] BARCELLOS, A. M. P. Projeto do Sistema Operacional de Rede Heterogêneo HetNOS. *In.*: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE (SEMISH), 13., 1993, Florianópolis, SC. **Anais...** Florianópolis: Sociedade Brasileira de Computação, 1993.

[BAA 95] BARCELLOS, A. M. P. **Um Sistema Operacional de Rede como Ferramenta de Apoio ao Desenvolvimento de Sistemas Distribuídos. Parte I** - Mecanismos e Modelos de Comunicação em Sistemas Distribuídos. Porto Alegre: CPGCC da UFRGS, 1995. (RP-254).

[BAA 95a] BARCELLOS, A. M. P. **Um Sistema Operacional de Rede como Ferramenta de Apoio ao Desenvolvimento de Sistemas Distribuídos. Parte II** - Programação Distribuída no Ambiente HetNOS. Porto Alegre: CPGCC da UFRGS, 1995. (RP-255).

[BAA 95b] BARCELLOS, A. M. P. **Um Sistema Operacional de Rede como Ferramenta de Apoio ao Desenvolvimento de Sistemas Distribuídos. Parte III** - Projeto e Implementação do Sistema HetNOS. Porto Alegre: CPGCC da UFRGS, 1995. (RP-256).

[BAB 85] BABAOGLU, Ö.; DRUMMOND, R. Streets of Byzantium: Netwotk Architectures for Fast Reliable Broadcast. **IEEE Transactions on Software Engeneering**, New York, v. SE-11, n. 6, p. 546-554, June 1985.

[BAB 86] BABAOGLU, Ö.; DRUMMOND, R.; STEPHENSON, P. The Impact of Communication Network Properties on Reliable Broadcast Protocols. *In*: FAULT TOLERANT COMPUTING SYSTEMS (FTCS), 16., 1986, Vienna, Austria. **Proceedings...** Washington, D.C.: IEEE, 1986. p. 212-217.

[BAB 87] BABAOGLU, Ö. On the Reliability of Consensus-Based Fault-Tolerant Distributed Computing Systems. **ACM Transactions on Computer Systems**, New York, v. 5, n. 4, p. 251, Nov. 1987.

[BAL 89] BAL, H.; STEINER, J.; TANENBAUM, A. Programming Languages for Distributed Computing Systems. **ACM Computing Surveys**, New York, v. 21, n. 3, Sept. 1989.

[BAM 93] BARBORAK, M.; MALEK, M.; DAHBURA, A. The Consensus Problem in Fault-Tolerant Computing. **ACM Computing Surveys**, New York, v. 25, n. 2, p. 171-220, June 1993.

[BAR 94] BARCELOS, P. P. A. **Estudo e Análise de Protocolos de Difusão Confiável**. Porto Alegre: CPGCC da UFRGS, 1994. (TI-370).

[BAR 95] BARCELOS, P. P. A.; WEBER, T. S. Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável. *In*: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS (SCTF), 6., Canela, RS. **Anais...** Porto Alegre: SBC, 1995.

[BEL 92] BELMONTE, V. R. F. **Gerência de Processos em Sistemas Distribuídos Tolerantes a Falhas.** Porto Alegre: CPGCC da UFRGS, 1992. Dissertação de Mestrado.

[BIR 87] BIRMAN, K.; JOSEPH, T. A. Reliable Communication in the Presence of Failures. **ACM Transactions on Computer Systems**, New York, v. 5, n. 1, p. 47-76, Feb. 1987.

[BIR 91] BIRMAN, K.; SCHIPER, A.; STEPHENSON, P. Lightweight Causal and Atomic Group Multicast. **ACM Transactions on Computer Systems**, New York, v. 9, n. 3, p. 272-314, Aug. 1991.

[BIR 93] BIRMAN, K. The Process Group Aproach to Reliable Distributed Computing. **Communications of the ACM**, New York, v. 36, n. 12, Dec. 1993.

[BUT 92] BUTLER, R. et al. **User's Guide to the p4 Programming System**. Argonne: ANL, 1992.

[CHA 84] CHANG, J.; MAXEMCHUK, N. F. Reliable Broadcast Protocols. **ACM Transactions on Computer Systems**, New York, v. 2, n. 3, p. 251-273, Aug. 1984.

[COU 94] COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems**: concepts and design. Workingham: Addison-Wesley, 1994.

[CRI 85] CRISTIAN, F. et. al. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *In*: FAULT TOLERANT COMPUTING SYSTEMS (FTCS), 15., 1985, Ann Arbor, USA. **Proceedings...** [S.l.: s.n.], 1985.

[CRI 91] CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, New York, v. 34, n. 2, p. 57-78, Feb. 1991.

[DRU 93] DRUMMOND, R.; BABAOGLU, Ö. Low-cost clock synchronization. **Distributed Computing**, Berlin, n. 6, p. 193-203, 1993.

[DUE 92] DUEÑAS, L. T. R. **Proposta e Estudo de um Algoritmo de Difusão Confiável para Aplicações Distribuídas Replicadas.** Florianópolis, SC: UFSC, 1992. Dissertação de Mestrado.

[FIS 85] FISCHER, M. J; LYNCH, N. A; PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. **Journal of the ACM**, New York, v. 32, n. 2, p. 374-382, Apr. 1985.

[FIS 86] FISCHER, M. J; LYNCH, N. A; MERRITT, M. Easy Impossibility Proofs for Distributed Consensus Problem. **Distributed Computing**, Berlin, n. 1, p. 26-39, Jan. 1986.

[GAR 82] GARCIA-MOLINA, H. Elections in a Distributed Computing System. **IEEE Transactions on Computers**, New York, v. 31, n. 1, p. 48-59, Jan. 1982.

[GEI 90] GEIST, A. et al. **PVM 3 User's Guide and Reference Manual**. [S.l.]: Oak Ridge National Laboratory, 1993.

[JAL 94] JALOTE, P. **Fault Tolerance in Distributed Systems**. Englewood Cliffs, New Jersey: Prentice-Hall, 1994.

[KAA 89] KAASHOEK, M. F. et. al. An Efficient Reliable Broadcast Protocol.

Operating Systems Review, New York, v. 23, n. 4, p. 5-20, Oct. 1989.

[LAM 78] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. **Communications of the ACM**, New York, v. 21, n. 7, p. 558-565, July 1978.

[LAM 82] LAMPORT, L.; SHOSTAK, R.; PEASE, M. The Byzantine Generals Problem. **ACM Transactions on Programming Languages and Systems**, New York, v. 4, n. 3, p. 382-401, July 1982.

[LAM 84] LAMPORT, L. Using Time Instead of Time-outs in Fault-Tolerant Systems. **ACM Transactions on Programming Languages and Systems**, New York, v. 6, n. 2, p. 256-280, Apr.1984.

[LAM 85] LAMPORT, L; MELLIAR-SMITH, P. Synchronizing Clocks in the Presence of Faults. **Journal of the ACM**, New York, v. 32, n. 1, Jan. 1985.

[LAP 85] LAPRIE, J. C. Dependable Computing and Fault Tolerance: Concepts and Terminology. *In*: FAULT TOLERANT COMPUTING SYSTEMS (FTCS), 15., 1985, Ann Arbor, USA. **Proceedings...** [S.l.: s.n.], 1985.

[MÓR 94] MÓRA, I. A. **Diagnóstico de Falhas em Sistemas Distribuídos**. Porto Alegre: CPGCC da UFRGS, 1994. Dissertação de Mestrado.

[PEA 80] PEASE, M; LAMPORT, L; SHOSTAK, R. Reaching Agreement in the Presence of Faults. **Journal of the ACM**, New York, v. 27, n. 2, p. 228-234, 1980.

[SCH 88] SCHMUCK, F. B. **The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems**. Ithaca, NY: Cornell University, Department of Computer Science, 1988. PhD Thesis.

[SEG 83] SEGALL, A; AWERBUCH, B. *A* Reliable Broadcast Protocol. **IEEE Transactions on Communications**, New York, v. COM-31, n. 7, p. 896, July 1983.

[SIL 91] SILBERSCHATZ, A.; PETERSON, J.; GALVIN, P. **Operating Systems Concepts**. Reading: Addison-Wesley, 1991.

[SIN 94] SINGHAL, M.; SHIVATATRI, N.G. **Advanced Concepts in Operating Systems**: Distributed, Database, and Multiprocessor

Operating Systems. New York: McGraw-Hill, 1994.

[SRI 87] SRIKANTH, T. K.; TOUEG, S. Optimmal clock synchronization. **Journal of the ACM**, New York, v. 34, n. 3, p. 626-645, July 1987.

[STE 90] STENES, W. R. **UNIX Network Programming**. Englewood Cliffs: Prentice-Hall, 1990.

[TAN 87] TANENBAUM, A. S. **Operating Systems Design and Implementation**. Englewood Cliffs: Prentice-Hall, 1987.

[TAN 89] TANENBAUM, A. S. **Computer Networks**. 2nd Ed. Englewood Cliffs: Prentice-Hall, 1989.

[TUR 92] TUREK, J.; SHASHA, D. The Many Faces of Consensus in Distributed Systems. **Computer**, Los Alamitos, v. 25, n. 6, June 1992.

[VER 89] VERÍSSIMO, P. J. E. **Comunicação em Grupo Fiável em Sistemas Distribuídos sobre Rede Local**. Lisboa: INESC, 1989. Tese de Doutorado.

[WAL 84] WALLECE, J. J.; BARNES, W. W. Designing for ultrahigh availability: the Unix RTR operating system. **Computer**, Los Alamitos, v. 17, n. 8, p. 31-39, Aug. 1984.

[WEB 90] WEBER, T. S; JANSCH-PORTO, I. E; WEBER, R. F. Fundamentos de Tolerância a Falhas. *In*: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA (JAI), 9., 1990, Vitória, ES. **Anais...** Vitória: SBC, 1990. 95p.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ADC - Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável.

por

Patrícia Pitthan de Araújo Barcelos

Dissertação apresentada aos Senhores:

Prof. Dr. Raimundo José de Araújo Macedo (UFBA)

Profa. Dra. Maria Lúcia Blanck Lisbôa

Prof. Dr. José Palazzo Moreira de Oliveira

Vista e permitida a impressão.
Porto Alegre, 27/08/96.

Profa. Dra. Taisy Silva Weber,
Orientador.

Prof. Flávio Rech Wagner
Coordenador do Curso de Pos Graduação
em Ciência da Computação - CPGCC
Instituto de Informática - UFRGS