Universidade Estadual de Campinas
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

UNICAMP

Algoritmos Distribuídos para Localização de Falhas e Difusão de Mensagens em Hipercubos Defeituosos

Autor: Saulo Rodrigues do Nascimento

Orientador: Prof. Dr. Marco A. Amaral Henriques

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos necessários para a obtenção do título de Mestre em Engenharia Elétrica.

Banca Examinadora:

Profa. Dra. Eliane Martins - IC/UNICAMP
Prof. Dr. Maurício Ferreira Magalhães - DCA/FEEC/UNICAMP
Prof. Dr. Marco A. Amaral Henriques - DCA/FEEC/UNICAMP

Dissertação de Mestrado

Campinas
Fevereiro, 2000

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

N17a

Nascimento, Saulo Rodrigues do
Algoritmos distribuídos para localização de falhas e difusão de mensagens em hipercubos defeituosos / Saulo Rodrigues do Nascimento.--Campinas, SP: [s.n.], 2000.

Orientador: Marco A. Amaral Henriques
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Tolerância a falha (Computação). 2. Hipercubo. I. Henriques, Marco A. Amaral. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Após investigação das soluções existentes na literatura para o problema da difusão de mensagens em máquinas paralelas tipo hipercubo com falhas de enlace e de nó, este trabalho faz uma análise detalhada das mesmas comparando os fatores que determinam qual é mais adequada para cada tipo de aplicação. Em seguida, efetua-se uma pesquisa sobre os métodos existentes para a detecção e localização de falhas em sistemas multicomputadores. Como resultado, constata-se a ausência de um método específico para hipercubos e propõe-se um algoritmo distribuído que explora as propriedades desta topologia na detecção e localização de enlaces e nós falhos. Finalmente, é proposto um algoritmo tolerante a falhas para difusão eficiente de mensagens, reunindo e superando as características positivas dos melhores algoritmos existentes. O algoritmo proposto tolera até $n-1$ falhas, sendo elas de enlaces, nós ou uma combinação de ambas, e realiza todo o processo de difusão em no máximo $n+2$ etapas, para um hipercubo de dimensão n. Um grande número de testes efetuados em um hipercubo comercial atesta a correção desta solução e a sua posição de vantagem em relação às existentes.

Abstract

In this work, it is made a detailed analysis of the solutions found in the literature for the problem of broadcasting in hypercube parallel machines with link and node failures, and all the aspects that determine which one fits better to a specific environment and application are compared. Following that, the existing methods used to detect and locate failures in multicomputer systems - major concern in a broadcasting process - are analyzed as well. Based on this research, it is realized the absence of a specific method for hypercubes and a new and distributed algorithm is proposed to fill in this gap. This algorithm takes advantage of the hypercube topology in order to detect and locate faulty links and nodes efficiently. Finally, it is proposed a new and efficient fault tolerant algorithm for broadcasting in hypercube systems, gathering and surpassing the most positive features of the former solutions. This algorithm treats up to n-1 failures (links, nodes or both) and finishes the whole process in no more than n+2 broadcasting steps, for an n-dimensional hypercube. Several tests performed on a commercial version of a hypercube-based machine confirms the correctness of this solution and its higher quality compared to the other ones.

Agradecimentos

Gostaria de agradecer aos meus pais Elson e Leolina pelos incansáveis esforços que sempre fizeram por mim e que me possibilitaram atingir metas cada vez mais altas. De grande importância também para conclusão deste trabalho foi o apoio que recebi de todos os amigos e pessoas com quem convivi estes últimos anos. Ao professor Marco Aurélio estarei eternamente grato pela paciência, compreensão e sábias orientações. E acima de todos eles, agradeço a Deus por ter me dado vida, por ter estado ao meu lado nos momentos mais difíceis e por ter me dado a compreensão de que sem Ele nada disso seria possível.

"O conhecimento é como asas para a vida do homem; é como uma escada pela qual ele possa ascender. Incumbe a cada um adquiri-lo. O conhecimento deve ser adquirido, porém, de tais ciências que possam prestar benefícios aos povos da terra, e não daquelas que por meras palavras começam e assim também terminam."

Bahá'u'lláh

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução Geral

Este estudo focaliza a questão da detecção e localização de falhas e da difusão de mensagens em hipercubos defeituosos. Apesar de serem dois assuntos distintos, eles estão estreitamente relacionados já que os algoritmos de difusão de mensagens, tolerantes a falhas, necessitam de um certo conhecimento da configuração de falhas do sistema para poderem funcionar.

1.1 Motivação

Sistemas distribuídos, multicomputadores e redes de interconexão são conceitos já bem firmados e sua utilização cresce rapidamente à medida que novas aplicações surgem e a demanda por capacidade computacional aumenta. O potencial destas tecnologias é grande, porém ele só se torna algo útil quando se desenvolvem pesquisas e experimentos para melhor entendê-lo e utilizá-lo de forma mais eficiente.

Em termos de topologia de rede para sistemas multicomputadores, o hipercubo binário (ou simplesmente, hipercubo) destaca-se por suas diversas características favoráveis; entre elas podemos citar alta conectividade entre os nós, o que possibilita maior tolerância a falhas, baixo diâmetro e fácil expansibilidade.

Os hipercubos são em geral encontrados em forma de sistema multicomputador local, ou seja, vários computadores (processadores + memória) interconectados e contidos em um mesmo gabinete. Porém, seu conceito tem sido ampliado e hoje já podemos encontrar os

chamados "hipercubos virtuais", que são máquinas interconectadas em rede local ou até mesmo através da Internet (como no caso do sistema JOIN [HUE 98]), que implementam os conceitos básicos e a topologia do hipercubo de forma lógica, aproveitando algumas de suas vantagens. Descrições de alguns sistemas que foram implementados com hipercubos podem ser encontradas em [NCU 90] e [HIL 85].

Em qualquer sistema composto de elementos interconectados, a ocorrência de falhas, seja de hardware ou de software, é algo inevitável. A eficiência e a exatidão de aplicações executadas em tais sistemas podem ser comprometidas no caso de ocorrência de falhas. Portanto, tolerância a falhas é um aspecto de alta relevância e por isso sempre atrai a atenção de pesquisadores. No sentido de tornar um sistema tolerante a falhas, acredita-se que o primeiro passo a ser dado é implementar no mesmo um algoritmo que detecte e localize as ocorrências de falhas eficientemente. Não foi encontrado na literatura nenhum de tais algoritmos que fosse específico para o hipercubo, tirando vantagem de sua estrutura. Uma das contribuições deste trabalho consiste em preencher esta lacuna com a proposta de um algoritmo de detecção e localização de falhas em hipercubos.

Dentre as várias atividades e operações que ocorrem em um sistema como o hipercubo, o processo de difusão de mensagens é realizado com bastante freqüência. De certa forma, a eficiência de uma aplicação depende do desempenho do algoritmo de difusão utilizado. A operação de difusão, como não poderia deixar de ser, é também prejudicada na presença de falhas, podendo se tornar mais lenta ou até mesmo não ser completada devidamente. Existem vários algoritmos propostos para difusão de mensagens em hipercubos falhos. Cada um destes algoritmos apresenta uma abordagem diferente no que diz respeito ao tratamento das falhas, cobrindo um determinado escopo do problema. Assim, eles podem ser mais adequados para alguns tipos de aplicações e ambientes específicos que outros. Apesar da maioria dos artigos encontrados na literatura sobre este assunto fazer menção a um ou outro algoritmo existente e propor um novo, nenhum deles faz uma compilação detalhada sobre o que já existe, destacando os pontos relevantes de cada solução e fazendo comparações.

Com o objetivo de se conhecer e avaliar o estado da arte no tocante à difusão de mensagens em hipercubos falhos, este trabalho provê uma compilação dos métodos existentes na literatura, acompanhada de uma comparação detalhada dos mesmos sob uma mesma referência. Feitas as comparações, constata-se a existência de situações de falha que não são tratadas adequadamente pelas técnicas comparadas. Procura-se então com este trabalho cobrir esta nova lacuna com a proposta de um novo algoritmo de difusão de mensagens tolerante a falhas que seja mais completo e eficiente que aqueles já existentes com a mesma finalidade.

1.2 Organização da dissertação

A dissertação está organizada como segue. Primeiramente, descreve-se no Cap. 2 os conceitos e definições utilizados neste trabalho, bem como considerações a respeito do hipercubo, da questão da difusão e das falhas. Nele, detalha-se o conceito de sistemas multicomputadores e se apresenta noções do hipercubo e de suas primeiras versões comerciais. A operação de difusão é explicada e são apresentados os métodos mais comuns de realizá-la, além de serem feitas considerações a respeito das difusões mono e multiporta, síncrona e assíncrona. Além disso, trata-se a questão das falhas diferenciando-se falhas de hardware de falhas de software, fixando-se o que será e o que não será tratado neste trabalho e apresentando-se a terminologia que será utilizada no mesmo. No Cap. 3, é apresentada uma análise e comparação de algoritmos de difusão de mensagens em hipercubos com falhas. São apresentados os aspectos básicos e característicos dos algoritmos de difusão tolerantes a falhas de forma que se tenha um ponto de partida para comparação. Cinco algoritmos mais representativos foram escolhidos para um estudo mais detalhado. As principais características de cada um destes algoritmos, incluindo suas limitações, são apresentadas numa tabela que facilita identificar qual deles é o mais adequado para um determinado ambiente ou aplicação. Análises de desempenho também são feitas e um conjunto de gráficos comparativos são apresentados mostrando o desempenho de cada algoritmo sob diferentes condições.

A questão da detecção e localização de falhas é tratada no Cap. 4. Neste capítulo, apresentam-se os aspectos mais relevantes sobre o assunto abordando-se os métodos mais tradicionais que se tem utilizado na detecção de falhas em sistemas multicomputadores. Estes métodos são analisados e suas vantagens e deficiências são destacadas. Os pontos principais dos algoritmos propostos mais recentemente para realizar tal tarefa são discutidos e percebe-se a falta da adequação dos mesmos, em algumas situações, quando utilizados no hipercubo, em especial no que se refere ao fornecimento de informações sobre falhas do sistema a outros algoritmos tais como os que implementam operações de difusão. Com a finalidade então de suprir esta deficiência, propõe-se ainda no Cap. 4 o algoritmo *DETECT* desenvolvido para realizar o processo de detecção e localização de falhas no hipercubo de forma eficiente. O método de funcionamento deste algoritmo, bem como os aspectos teóricos que dão base à sua proposição, são apresentados através de lemas, teoremas, exemplos e figuras. A parte final do capítulo é reservada para a análise de desempenho do algoritmo proposto, dando-se detalhes da sua implementação em um hipercubo comercial e apresentando-se gráficos e tabelas gerados a partir dos dados colhidos durante os testes.

O algoritmo *ESLAB*, proposto para substituir os algoritmos atuais de difusão de mensagens em hipercubos defeituosos, é então descrito no Cap. 5. Deixou-se para apresentar o *ESLAB* somente após a discussão sobre os aspectos da detecção e localização de falhas pois ele utiliza o algoritmo *DETECT* para lhe fornecer as informações necessárias sobre a configuração de falhas do sistema, e a forma como ele utiliza estas informações é uma parte fundamental de sua estrutura. Assim como no caso do algoritmo *DETECT*, apresenta-se lemas, teoremas, exemplos e figuras para facilitar o entendimento do funcionamento de *ESLAB*. Vários testes foram realizados em um hipercubo comercial com este e outros dois algoritmos de difusão de mensagens. Os resultados destes testes e das comparações feitas entre os algoritmos, bem como considerações sobre aplicações para *ESLAB*, são também apresentados no Cap. 5.

Finalmente, as conclusões finais e uma discussão sobre possíveis trabalhos futuros são apresentadas no Cap. 6. Neste capítulo faz-se uma revisão das principais contribuições deste trabalho e lista-se as publicações e palestras que resultaram do mesmo. A seção com as referências bibliográficas aparece logo a seguir. Na parte dos Apêndices, é listado o pseudo-código dos algoritmos *DETECT*, *ESLAB* e dos outros dois que foram implementados e testados em um hipercubo real. Algumas considerações relacionadas à questão dos testes e das dificuldades encontradas na medição de desempenho são descritas também na parte dos Apêndices.

Capítulo 2

Conceitos e Definições

Neste capítulo, apresentaremos conceitos e definições utilizados nesta dissertação. Com o intuito de facilitar o entendimento do que é tratado na mesma e evitar duplicidade de significados, é apresentada aqui uma descrição dos objetos de estudo e a terminologia básica utilizada com os mesmos.

2.1 Hipercubos

Com o surgimento dos microprocessadores no começo dos anos 70, máquinas de computação razoavelmente baratas e poderosas se tornaram disponíveis. Poucos anos mais tarde surgiram os sistemas multicomputadores. Estes sistemas foram concebidos como computadores paralelos de propósito geral, sem compartilhamento de memória e tipicamente programados usando linguagens de programação seqüencial convencionais. Em geral, o atrativo dos multicomputadores é baseado na exploração efetiva da tecnologia VLSI, nos altos graus de paralelismo que com eles se consegue e no preço moderado de seu hardware [REE 87].

Um dos primeiros projetos de redes multicomputadores, o Columbia Homogeneous Parallel Processor [SUL 77], era baseado na topologia hipercúbica binária. As vantagens oferecidas por esta topologia, detalhada a seguir, foram rapidamente percebidas e um grande interesse pela mesma surgiu, como ocorreu com o Cosmic Cube [SEI 85], protótipo desenvolvido no Instituto de Tecnologia da Califórnia. As primeiras versões comerciais do

hipercubo[1] começaram então a surgir, sendo a primeira delas o iPSC (intel Personal Super Computer) da Intel [RAT 85], em 1985.

2.1.1 Principais características

O hipercubo binário, também chamado de n-cubo binário ou simplesmente n-cubo, é caracterizado por um único parâmetro denominado **dimensão do hipercubo**. Este parâmetro determina o número de **nós** (processadores), o número de **enlaces de comunicação** conectados a cada nó e o **diâmetro** do hipercubo (menor distância entre os dois nós mais distantes). Um hipercubo n-dimensional, que pode ser mapeado em um sistema de coordenadas n-dimensional, é composto de 2^n nós, cada um dos quais ligado por enlaces de comunicação a n outros, um em cada dimensão, totalizando $n2^{n-1}$ enlaces em todo o sistema. O diâmetro do hipercubo é igual ao número de dimensões que ele possui. Um hipercubo de n+1 dimensões pode ser construído a partir de dois outros de dimensão n, conectando-se por enlace de comunicação cada nó de um com o seu correspondente no outro.

Cada nó do hipercubo possui, além do processador, sua própria memória local e sistema de comunicação com outros n nós.

2.1.2 Representação

Cada nó de um n-cubo é representado por uma tupla de n dígitos binários. Dois nós conectados por um enlace são chamados de vizinhos e diferem em seus rótulos em apenas um bit. Os enlaces são rotulados de forma similar ao nós, porém, apresentam um $*$ na posição do rótulo que indica a dimensão pela qual ele conecta os dois nós. A Fig. 2.1 exibe um hipercubo de 3 dimensões com a representação dos elementos básicos que aparecerão ao longo desta dissertação. No canto superior direito da figura está representado o sistema de coordenadas tri-dimensional utilizado na rotulação dos nós e enlaces. Nota-se, por exemplo, através desta representação, que o enlace $00*$ e o nó 111 são elementos falhos ou defeituosos.

Denomina-se subcubo a uma parte qualquer do hipercubo que constitui um hipercubo de menor dimensão. Os subcubos são representados com o valor fixo (0 ou 1) nos rótulos dos

[1] O termo hipercubo é usado doravante para designar rede multicomputador baseada na topologia hipercúbica binária.

nós que ele engloba, nas dimensões que ele não possui, e $*$'s em todas as outras dimensões. O subcubo formado pelos nós 000, 010, 100 e 110 da Fig. 2.1, por exemplo, é representado como $**0$.

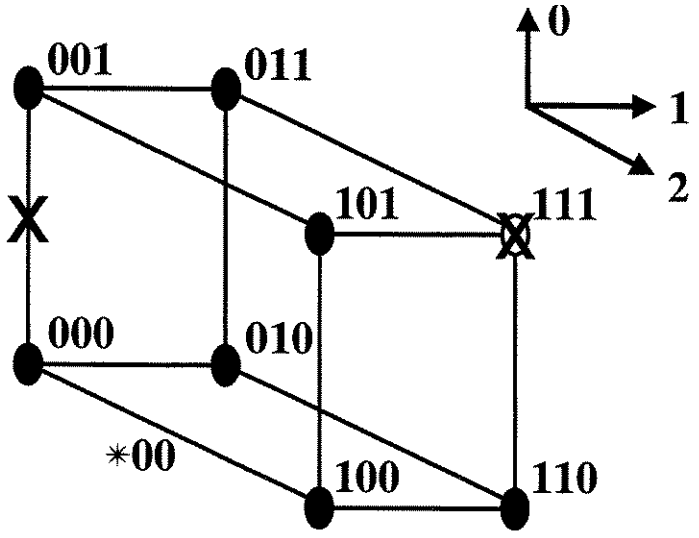

Figura 2.1: Exemplo de hipercubo de 3 dimensões e com componentes falhos.

2.1.3 Vantagens e desvantagens dos hipercubos

O hipercubo foi amplamente difundido devido a uma série de vantagens que oferece. Entre elas podemos citar a compatibilidade desta topologia com os padrões de comunicação das tarefas criadas a partir da decomposição natural de muitos algoritmos numéricos. Sua rica estrutura de interconexão comporta em si a simulação de importantes topologias como a linear, a de anel, a de grade n-bidimensional, entre outras. Essa mesma estrutura de interconexão permite que o diâmetro do hipercubo cresça de forma logarítmica com o número de nós. Outra vantagem está na facilidade de expansão física oferecida pela característica recursiva que esta topologia tem. Módulos contendo um hipercubo k-dimensional podem ser usados na construção de hipercubos de dimensões maiores. Além disso, o alto grau de conectividade entre os nós torna o hipercubo tolerante a falhas, já que existem vários caminhos de comunicação entre dois nós quaisquer do mesmo.

Apesar destas vantagens, o hipercubo apresenta duas principais deficiências. Como um hipercubo de k dimensões é formado por dois outros de k-1 dimensões, percebe-se que só

é possível expandir um hipercubo dobrando-se seu número de processadores. Outro ponto é que o número de conexões por nó cresce com o número de dimensões. Assim sendo, o número máximo de dimensões deve ser determinado em tempo de projeto de hardware já que é neste momento que o número de conexões possíveis por nó é fixado. Poder-se-ia pensar em projetar os nós com um número bem grande de canais de entrada e saída para que, em caso de necessidade, se pudesse expandir o hipercubo livremente, porém isto poderia acarretar em um aumento excessivo do custo do projeto e/ou atingir um limite tecnológico, já que o número máximo possível de conexões por nó não é ilimitado.

2.2 A Operação de difusão

Em uma rede de interconexão, difundir uma mensagem significa fazer com que todos os pontos da rede, ou seja, todos os processadores, recebam esta mensagem. O nó de onde a mensagem é originada é chamado **nó origem** ou **nó fonte**. Basicamente, a difusão consiste em que, uma vez que o nó fonte dá início ao processo, cada nó que recebe a mensagem retransmite a mesma para seus vizinhos obedecendo a lógica determinada pelo método utilizado. Existem várias formas de se realizar tal tarefa, porém busca-se sempre fazê-la da maneira mais rápida e econômica em termos de utilização de recursos. A forma mais rápida de difundir uma mensagem é fazer com que ela atinja cada nó, a partir do nó origem, percorrendo o menor caminho existente entre os dois, de forma que o número de etapas de execução do processo seja igual ao diâmetro da rede. Por exemplo, no caso de um n-cubo, a forma mais rápida de difundir uma mensagem deve usar n etapas de execução, já que n é o diâmetro do cubo. A forma mais econômica seria aquela pela qual cada nó só recebe a mensagem uma única vez. O fato de um nó receber uma mesma mensagem mais de uma vez caracteriza uma **redundância**.

Os dois métodos tradicionais de difusão de mensagens em hipercubos são representados na Fig. 2.2. A **difusão multiporta** é aquela em que cada nó envia a mensagem ao seus vizinhos de forma simultânea. Obviamente este tipo de difusão só é possível para hipercubos cujos circuitos de rede e software de comunicação suportam este tipo de comunicação. Num modelo ideal, desconsiderando-se os atrasos de armazenamento e comunicação, cada etapa do processo marca o exato momento em que os nós recebem a mensagem, assim como mostra a Fig. 2.2(a). Na **difusão monoporta** por outro lado, cada nó envia a mensagem apenas para um vizinho por etapa. Aqui também, num modelo ideal como o mostrado

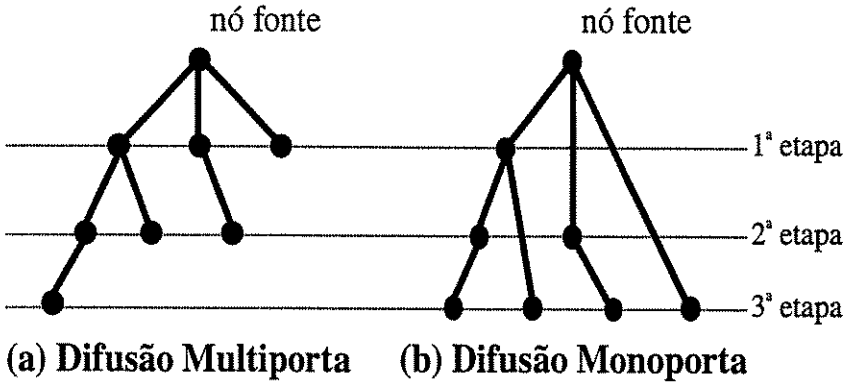

Figura 2.2: Difusão de mensagem em um hipercubo de 3 dimensões.

na Fig. 2.2(b), cada etapa do processo está relacionada com o exato momento em que os nós recebem a mensagem difundida. Tanto na difusão multiporta quanto na monoporta, pode-se ter **comunicação síncrona**, quando um processo espera a confirmação de recebimento da mensagem por parte do nó destino para continuar executando suas tarefas, ou **comunicação assíncrona**, quando nenhuma confirmação é esperada. Um processo de difusão que ocorre em um ambiente onde a comunicação é assíncrona, é mais rápido porém pode não ser tão confiável quanto em um ambiente onde a comunicação é síncrona.

O conjunto de caminhos que a mensagem percorre dentro do hipercubo desde o nó origem até os nós mais distantes é denominado de **árvore de difusão**, sendo o nó origem a raiz desta árvore. Cada nível na árvore de difusão caracteriza uma etapa de execução do processo. Assim, como pode-se ver na Fig. 2.2, os dois métodos gastam exatamente três etapas para realizar o processo de difusão em um hipercubo tri-dimensional (8 nós).

Até o momento, nos referimos à difusão de mensagens apenas em sistemas não defeituosos. Na presença de falhas o processo de difusão de uma mensagem passa a ser o de, a partir de um nó origem, atingir todos os nós não-defeituosos do sistema. Os métodos de difusão de mensagens devem ser alterados de forma a contornar as falhas que possam existir.

2.3 Falhas em sistemas multicomputadores

Sabe-se que quanto maior o número de componentes interconectados em um sistema, maior é a probabilidade de ocorrência de falhas. Vários são os artigos encontrados na literatura que tratam com profundidade a questão das falhas em redes multicomputadores. Nesta seção, são apresentados apenas os pontos básicos indicando que tipos de falhas estarão sendo tratados ao longo da dissertação.

As falhas, em geral, se dividem em dois grupos:

FALHAS DE HARDWARE

São aquelas relacionadas com a parte física do equipamento. Em um sistema multicomputador como o hipercubo, a complexidade do hardware decorrente da interconexão de componentes possibilita o surgimento de diversos pontos passíveis de defeito, começando pelos próprios processadores que podem apresentar defeito de fabricação, comportamento inesperado por decorrência de sobreaquecimento, pinos danificados, entre outras causas. Existe também uma série de problemas que podem ocorrer por conta da memória local. Além disso, um nó pode estar funcionando perfeitamente, mas defeitos em seu circuito de entrada e saída podem deixá-lo completamente inutilizável.

Além destes, que podem ser considerados defeitos isolados, existem aqueles que surgem por decorrência da própria integração. Por exemplo, a manutenção dada em uma parte do equipamento pode ter efeitos colaterais em outras partes do sistema. Estes são apenas alguns exemplos, podendo-se ainda enumerar uma série de outros defeitos relacionados ao hardware.

FALHAS DE SOFTWARE

São aquelas relacionadas à parte lógica do sistema. Em geral, todo defeito que não for de hardware é enquadrado como sendo de software. Diz respeito a erros por parte do sistema operacional, software de roteamento, entre outros. Este tipo de falha pode comprometer o funcionamento de todo o sistema e é em geral difícil de ser detectado e corrigido [DEE 99].

Nesta dissertação serão considerados apenas dois tipos de falhas. São eles:

- Falha em Nó: quando um nó permanece incomunicável com qualquer um de seus vizinhos, seja qual for o motivo;
- Falha em Enlace: quando um nó permanece incomunicável com um vizinho específico através do enlace que os conecta, seja uni ou bidirecionalmente, e seja qual for o motivo.

Para facilitar o entendimento de certos lemas e teoremas que serão apresentados ao longo do texto, tem-se associadas a estes dois tipos de falhas as seguintes definições.

- Definição 1: *enlace falho* é um enlace que não consegue transmitir dados em uma ou nenhuma das duas direções.
- Definição 2: *enlace não-falho* é um enlace que consegue transmitir dados em ambas as direções.
- Definição 3: *nó falho* é um nó que não consegue se comunicar com nenhum outro nó.
- Definição 4: *nó não-falho* é um nó que consegue se comunicar com pelo menos um outro nó.
- Definição 5: *nó perfeito* é um nó não-falho que não possui enlaces falhos.
- Definição 6: *nó não-perfeito* é um nó não-falho com pelo menos um enlace falho.

As falhas também são categorizadas como sendo permanentes ou intermitentes. As falhas permanentes são aquelas que uma vez detectadas vão continuar presentes no sistema até que sejam devidamente corrigidas. As intermitentes, por outro lado, surgem e desaparecem sem que haja a interferência de um operador. Elas podem surgir durante a execução de uma aplicação, comprometendo os resultados obtidos, e desaparecer antes que a aplicação termine, passando despercebidas pelo usuário. Serão tratadas neste trabalho apenas as falhas permanentes.

Embora a questão das falhas seja um dos principais aspectos do estudo realizado, está fora do escopo do mesmo a análise da freqüência com que estas ocorrem e o motivo pelo qual elas aparecem em um sistema.

Capítulo 3

Algoritmos de Difusão de Mensagens

Denomina-se **algoritmos de difusão** os algoritmos que implementam métodos de difusão de mensagens. Especificamente neste capítulo estaremos analisando apenas os algoritmos de difusão tolerantes a falhas, ou seja, aqueles que são capazes de realizar com sucesso uma operação de difusão de mensagem mesmo na presença de falhas no sistema.

3.1 Principais características dos algoritmos de difusão

Um algoritmo de difusão pode tolerar apenas nós falhos, apenas enlaces falhos ou uma combinação de ambos. Alguns algoritmos tratam apenas enlaces falhos e simplificam nós falhos como sendo aqueles que possuem todos os seus enlaces falhos. O número de falhas no sistema é sempre algo imprevisível e uma limitação para os algoritmos. Assim sendo, todo algoritmo de difusão tolerante a falhas especifica o número máximo de falhas que ele tolera para ainda garantir sucesso na operação de difusão. Além disso, alguns algoritmos especificam também o número máximo de falhas que podem estar presentes no sistema e ainda haver a chance de se realizar a difusão com sucesso, dependendo da distribuição das falhas. Fica claro que, quanto maior o número e os tipos de falhas toleradas, melhor é o algoritmo.

Uma questão importante relacionada com a distribuição das falhas pelo sistema é identificar que tipo de informação de configuração de falhas os nós precisarão armazenar para executar corretamente os algoritmos. Esta questão tem um impacto direto no desempenho

dos algoritmos e na utilização dos recursos de hardware que eles demandam, assim como será visto no Cap. 4. Dependendo da estratégia utilizada pelo algoritmo de difusão, os nós precisarão apenas da informação da configuração de falhas de sua vizinhança direta, ou seja, daqueles nós a 1 passo, ou 1 dimensão, de distância. Neste caso, diz-se que os nós necessitam **informação local**. Quando os nós precisam de informação a respeito de vizinhos a k (onde $k <$ diâmetro da rede) dimensões distante deles, eles armazenam **informação global limitada**. **Informação global** é necessária quando os nós precisam conhecer a distribuição das falhas por todo o sistema. Deduz-se facilmente que, quanto menor a quantidade de informação sobre as falhas os nós precisarem obter, melhor.

Outra característica de tais algoritmos está relacionada com o nó origem. Eles podem permitir que qualquer nó do sistema inicie o processo de difusão ou que apenas alguns nós específicos o possam fazer. Mesmo que algumas aplicações venham a necessitar que apenas um subgrupo de nós difunda mensagens, é sempre conveniente ter a possibilidade de difundir a partir de qualquer nó.

O desempenho do algoritmo, ou seja, a velocidade com que executa a operação de difusão, é outra importante característica desses algoritmos, além do tratamento do *deadlock* e das redundâncias. Em resumo, os fatores que caracterizam os algoritmos de difusão de mensagens tolerantes a falhas podem ser resumidos como segue;

- tipos de falhas tolerados: somente nós, somente enlaces ou uma combinação de ambos;
- número máximo de falhas tolerado;
- tipo de estratégia utilizada;
- definição dos nós que podem iniciar um processo de difusão;
- tipo de informação de configuração de falhas que os nós necessitam;
- velocidade da operação;
- tratamento dado à questão do *deadlock* e das redundâncias.

3.2 Revisão bibliográfica

Cinco algoritmos de difusão de mensagens, tolerantes a falhas, em hipercubos foram estudados mais profundamente por apresentarem características interessantes ou soluções

inovadoras. Uma subseção apresentando um sumário de suas principais características é dedicada para cada um deles. Outros algoritmos também foram estudados, porém não foram tratados para efeito de comparação. Para estes últimos, foi dedicada uma subseção única para apresentar um resumo de suas principais características e o motivo pelo qual não foram incluídos na comparação.

A idéia principal é identificar o que há de melhor em cada algoritmo, analisando as soluções apresentadas e comparando cada uma delas com as outras em todos os aspectos relevantes. Assim, é possível ter uma noção geral de qual algoritmo é o mais adequado para cada tipo de aplicação e ambiente, e também se cria uma base para estudos que levem à criação de algoritmos novos e mais eficientes. O algoritmo *ESLAB* proposto no capítulo 5 é fruto deste estudo.

3.2.1 Metodologia de comparação

Alguns dos trabalhos pesquisados propõem mais de um algoritmo. Eles também fazem algumas suposições e apresentam variações no número de etapas necessárias para completar a difusão e no desempenho esperado de acordo com os tipos e a quantidade de falhas no sistema. No sentindo de colocá-los sob uma mesma referência, são feitas algumas considerações e suposições para que se consiga uma comparação justa. Por exemplo, o custo total exato de tempo para cada algoritmo completar a difusão varia de acordo com a implementação. Por simplicidade, considera-se que todos foram implementados em uma mesma linguagem e executados numa mesma máquina, isto é, avaliados sob as mesmas condições. Durante esta fase, nenhum algoritmo foi implementado e executado em um hipercubo real. Os custos de desempenho apresentados são estimativas baseadas na análise detalhada dos códigos e/ou descrições dos algoritmos apresentados nos artigos. Estes custos baseiam-se em número de **passos**, onde cada *passo* é uma operação no código do programa, como uma atribuição ou operação aritmética. Assume-se que os custos de enviar e receber uma mensagem são respectivamente de 2 *passos* e 3 *passos*, baseando-se no fato de que o custo do envio de uma mensagem inclui apenas a escrita da mensagem numa área de memória onde o sistema operacional possa apanhá-la e enviá-la sem afetar o desempenho do algoritmo. O custo de recebimento de uma mensagem inclui a retirada da mesma do buffer de leitura e o seu armazenamento numa área de memória utilizada pelo algoritmo.

Em alguns pontos no processo de estimativa dos custos de desempenho dos algoritmos (como o número de falhas presentes no hipercubo ou o número total de etapas para

completar a difusão, por exemplo) considera-se sempre o pior caso. Em outros, como no caso do tempo gasto na busca de um elemento em um vetor, considera-se o tempo médio entre o melhor e o pior caso. O importante a se notar é que foi usada a mesma política para todos os algoritmos. Para cada um dos cinco algoritmos comparados, é apresentado o custo total do esforço inicial e o custo para cada passo da difusão em si. O esforço inicial é aquele que é feito para preparar os nós para estarem aptos a iniciar uma difusão, como por exemplo recolher a informação da configuração de falhas necessária. Este esforço inicial, dependendo da solução, é necessário sempre que uma difusão é realizada ou quando for detectada uma mudança no sistema.

A seguir, são apresentados resumos das principais características dos algoritmos pesquisados, obedecendo a ordem cronológica de sua proposição.

3.2.2 Lee & Hayes - 1992

No artigo [LEE 92] T. C. Lee e J. P. Hayes propõem dois algoritmos; porém considera-se aqui apenas o que permite que qualquer nó não falho inicie um processo de difusão.

Os autores apresentam o conceito dos nós não-seguros (*unsafe nodes*) como estratégia utilizada pelo algoritmo. Basicamente, um nó é conceituado como sendo ativo, não-seguro, ou falho. Um nó é considerado **não-seguro** se possuir dois ou mais vizinhos falhos ou não-seguros, ou for não-perfeito. Nós não-falhos que não são não-seguros são denominados **ativos**. Quando o nó origem é ativo, o algoritmo completa a difusão em um n-cubo em n etapas. Quando o nó origem é não-seguro, ele primeiro envia a mensagem a um vizinho ativo. Este vizinho então inicia a difusão garantindo que a mensagem não será reenviada ao nó origem inicial. Nesta situação, a difusão é completada em $n + 1$ etapas.

O algoritmo assume que a princípio todos os nós sabem se são perfeitos ou não-perfeitos e conhecem os vizinhos que são falhos. O custo de determinar o *status* de todos os nós só é considerado uma vez, não importa quantas operações de difusão são realizadas, já que uma vez determinado ele é armazenado e permanece válido enquanto a máquina estiver ligada ou enquanto o algoritmo de determinação deste *status* não for reexecutado.

Apesar deste algoritmo tratar falhas de nó e de enlace, só é levado em consideração no artigo o número máximo de nós falhos tolerados ($\lceil \frac{n}{2} \rceil$) como condição suficiente para completar uma difusão com sucesso. Mais que $\lceil \frac{n}{2} \rceil$ nós falhos podem ser tolerados desde que sua distribuição no sistema não torne o hipercubo completamente não-seguro, ou seja, com todos os nós com *status* de não-seguros.

Os custos determinados a partir de uma métrica única usando as suposições em 3.2.1 são os seguintes.

- Definição do *status*: $2n^3 + 5n^2 + 2n$ *passos*
- Cada etapa na difusão: $\frac{n^2}{2} + 8n + 5$ *passos*
- Etapa extra caso o nó origem não seja ativo: $n + 4$ *passos*

3.2.3 Joshua Bruck - 1994

No algoritmo apresentado na referência [BRU 94] somente enlaces falhos são tolerados. O conceito das árvores disjuntas (*edge-disjoint trees* [JOH 89]) é a estratégia utilizada neste caso. Estas árvores disjuntas são um subconjunto das árvores binomiais onde cada enlace está presente em no máximo duas árvores. As árvores binomiais são aquelas formadas a partir de uma difusão em que cada nó difunde a mensagem em todas as dimensões exceto por aquelas já percorridas pela mesma em seu percurso desde o nó origem até aquele nó. Considerando-se que cada enlace pertence a no máximo duas árvores disjuntas, haverá sempre pelo menos uma **árvore sadia**, ou seja, uma árvore sem ramos (enlaces) falhos, a ser escolhida por qualquer nó fonte que deseja iniciar um processo de difusão, quando o número de enlaces falhos no hipercubo é no máximo $\lceil \frac{n}{2} \rceil - 1$. Dependendo da configuração de falhas do sistema, o algoritmo pode tolerar até $n - 2$ enlaces falhos.

O número de etapas necessárias para se completar o processo de difusão é sempre o ótimo, n. Assume-se a existência de um nó gerenciador que conhece a configuração de falhas de todo o sistema e pode difundir esta informação a todos os outros nós do hipercubo.

Abaixo são apresentados os custos determinados a partir de uma métrica única utilizando as suposições em 3.2.1.

- Construção das árvores disjuntas + escolha de uma sadia: $2^n n(3n - \frac{3}{4}) + (\frac{5}{2})(n^2 - n - 1)$ *passos*
- Cada etapa na difusão: $2^{n-1} + 9$ *passos*

3.2.4 Jie Wu - 1994

No artigo [JWU 94] publicado por Jie Wu, o processo chamado de *splitting* é utilizado para encontrar uma árvore binária de difusão sadia a partir do nó fonte. Este processo divide o subcubo destino, ou seja, aquele que deve ser atingindo a partir de um nó específico, em um grupo de subcubos destino disjuntos, em ordem crescente de dimensões, de forma a isolar o maior número de falhas possível no subcubo de maior dimensão. Este processo de divisão é orientado por uma seqüência especial de dimensões obtida a partir da análise da configuração de falhas do sistema. Assume-se que todos os nós do hipercubo conhecem a configuração de falhas de todo o sistema. Diversos algoritmos são propostos neste artigo, porém apenas é considerado aqui aquele que é ótimo no sentido de que ele sempre encontra uma árvore de difusão binária, caso exista, a partir do nó fonte. Este algoritmo, no entanto, só tolera enlaces falhos, sendo $n-1$ o número máximo de enlaces tolerados para se garantir o resultado esperado, e $n2^{n-1}-2^n$ é o número de falhas a partir do qual o algoritmo não é mais aplicável. Se o processo de *splitting* for ótimo, isto é, conseguir isolar todas as falhas no subcubo de maior dimensão, o número de etapas para completar o processo de difusão será também ótimo (n). De outra forma, o algoritmo precisa de $n+1$ etapas para finalizar a difusão. Considera-se sempre o pior caso para efeito de comparação.

Mostra-se a seguir os custos determinados a partir de uma métrica única baseada nas suposições em 3.2.1.

- Execução do processo de *splitting*: $4n^3+2n^2$ *passos*
- Cada etapa na difusão: $n+12$ *passos*

3.2.5 Jie Wu - 1995

O algoritmo proposto por J. Wu em [JWU 95] trata apenas de ocorrências de nós falhos no sistema. A estratégia utilizada baseia-se no conceito dos níveis de segurança (*safety levels*), explicado em detalhes na Seção 5.2. Os níveis de segurança guiam o processo de difusão de forma que, em cada etapa, os nós difusores (aqueles que já receberam a mensagem e a estão difundindo na etapa corrente) escolhem a seqüência de dimensões pela qual difundir a mensagem de acordo com quão confiáveis os vizinhos são para receber a mensagem e difundí-la para os nós que ainda não a receberam, até que todo o processo termine. Os nós com maior nível de segurança são escolhidos primeiro.

Mostra-se que na presença de até $n-1$ nós falhos em um n-cubo, todos os nós não-falhos recebem a mensagem difundida em exatas n etapas quando o nó origem possui o grau mais alto de segurança (nível de segurança n). Caso contrário, o processo é finalizado em $n+1$ etapas. Os nós só necessitam de informação local sobre as falhas.

Os custos determinados a partir de uma métrica única usando as suposições em 3.2.1 são mostrados abaixo.

- Definição dos níveis de segurança: $n^3 + n^2 + 4n - 6$ *passos*
- Cada etapa na difusão: $4n + 6$ *passos*

3.2.6 Jie Wu - 1996

O algoritmo apresentado em [JWU 96] baseia-se na estratégia denominada *extended spanning binomial tree structure* para encontrar uma árvore binária de difusão que seja sadia. Esta solução garante que o processo de difusão sempre será finalizado em exatas n etapas para um hipercubo de n dimensões quando o número de enlaces falhos não exceder $n-2$. Apenas enlaces falhos são considerados por este algoritmo.

Cada nó deve armazenar informação global limitada da configuração de falhas do sistema. A complexidade de tempo para coletar esta informação, entretanto, não é considerada no artigo e tampouco será considerada aqui já que não se tem parâmetros suficientes.

Abaixo são exibidos os custos determinados a partir de uma métrica única usando as suposições em 3.2.1.

- Custos iniciais: $n^2 + 5n$ *passos*
- Cada etapa na difusão: $\frac{n^3}{8} + \frac{21}{8}n^2 + 2n + 5$ *passos*

3.2.7 Outros algoritmos

O artigo "Broadcasting in Hypercubes with Link/Node Failures"[PAR 92] escrito por S. Park e B. Bose em 1992, ao contrário do que se pode concluir a partir do título, propõe

dois algoritmos, sendo um que suporta enlaces falhos e outro que suporta nós falhos. Nenhum algoritmo que suporte os dois ao mesmo tempo é apresentado. Os autores adotam uma estratégia similar ao do processo de *splitting*, descrito anteriormente, e partem do suposto que todos os nós conhecem a localização de todas as falhas no hipercubo. Para até $n-1$ falhas presentes no sistema, são necessárias $n+1$ etapas no processo de difusão. Para um número de falhas maior (até $2n-3$) são necessárias $n+3$ etapas. Nenhum dos algoritmos propostos neste artigo foram incluídos na comparação porque utilizam uma técnica similar a de um outro proposto mais recentemente (o de Jie Wu 1994), apresentam um número de etapas necessárias para completar a difusão igual ou maior que a dos outros analisados e necessitam de informação global sobre as falhas.

No artigo "A Fault-Tolerant Tree Communication Scheme for Hipercube Systems" [LEU 96] apresentado por Yuh-Rong Leu e Sy-Yen Kuo em 1996, é proposto um algoritmo que, apesar de não ser específico para difusão, pode ser facilmente modificado para tal. Na realidade ele é dirigido para resolver o problema de *merge* em hipercubos com enlaces falhos. A estratégia utilizada é a de dividir a solução em uma fase estática e outra dinâmica. A fase estática compreende encontrar uma árvore de difusão com o menor número possível de enlaces falhos. Caso não seja possível encontrar uma árvore de difusão sem enlaces falhos, a fase dinâmica se encarrega de fazer o roteamento da mensagem para os nós que não a puderem receber. A árvore de difusão é definida pelo nó origem e pela ordem das dimensões a serem percorridas durante o processo. O nó origem deve ser um nó que não apresenta enlaces falhos e a ordem das dimensões é definida partindo-se dele sempre em direção aos vizinhos que possuem uma vizinhança com menor número possível de enlaces falhos. A complexidade do algoritmo para definir a árvore de comunicação é $O(2^n)$, onde n é a dimensão do hipercubo. São necessárias $(n+k)$ etapas para completar a operação, onde k é o número de estágios onde há a necessidade de reroteamento. O número de enlaces falhos deve ser menor que 2^{n-1} para garantir a existência de pelo menos um nó que não apresente enlaces falhos. Este algoritmo não foi incluído na comparação pelo fato de não ter sido proposto como algoritmo de difusão de mensagens, e se fosse alterado para poder realizar operações de difusão, não seria mais o mesmo algoritmo assim como concebido pelos seus autores. A idéia de fazer com que cada nó monte um vetor de falhas que indique a situação de falha em cada uma de suas dimensões e depois compartilhe este vetor com os seus vizinhos, foi aproveitada e adaptada na elaboração do algoritmo *ESLAB* apresentado no Cap. 5.

3.2.8 Análises e comparações

Nesta seção será apresentada uma análise comparativa dos cinco primeiros algoritmos através de uma tabela geral de comparação e dos dados conseguidos através da estimativa do desempenho sob diferentes circunstâncias.

A Tabela 3.1 exibe nas linhas as principais características de algoritmos para difusão em hipercubos defeituosos e, nas colunas, os cinco algoritmos analisados detalhadamente. Esta tabela tem o objetivo de facilitar a escolha do algoritmo que é mais adequado para um determinado ambiente ou aplicação. Por exemplo, suponha-se um sistema com muitos nós falhos e sem acesso à informação global de falhas. A partir da Tabela 3.1, é fácil perceber que o Algoritmo de Jie Wu (1995) é a melhor opção para tal sistema. Entretanto, se o sistema também apresenta problemas de enlaces falhos e o número total de componentes falhos é pequeno, então o algoritmo de Lee & Hayes (1992) é a melhor escolha. As melhores opções em cada linha da tabela estão sublinhadas e em negrito para facilitar a identificação.

Com relação ao desempenho, alguns dos algoritmos analisados necessitam de informação sobre a configuração de falhas do sistema, mas não indicam os custos de desempenho que a obtenção desta informação acarreta. Apesar destes custos não poderem ser desprezados, não há como estimá-los já que eles podem depender da implementação do hardware. Assim, eles não serão levados em conta para efeito de comparação de desempenho mas contarão como ponto negativo para os algoritmos que os apresentam.

A seguir, são apresentados alguns gráficos que permitem visualizar o relacionamento entre o desempenho dos algoritmos, a dimensão do hipercubo e o número de difusões consecutivas que são realizadas. Estes dois últimos parâmetros são as variáveis de controle, uma vez que se está considerando o número de falhas no sistema como sendo o máximo que cada algoritmo pode tolerar para garantir os resultados desejados. Vale notar que alguns algoritmos, como o de J. Bruck (1994), necessitam realizar o processo de inicialização apenas para a primeira difusão, pois o que é feito neste estágio não precisa ser refeito em difusões subsequentes. Outros algoritmos, como o de Jie Wu (1994), ao contrário, necessitam realizar o processo de inicialização para toda difusão. Este fato é a principal razão para as variações de desempenho observadas nos gráficos.

Fica claro a partir dos dados colhidos que o algoritmo de Jie Wu (1995) apresenta o melhor desempenho em todas as situações quando comparado com todos os outros algoritmos. Isto ocorre por ele tratar apenas nós falhos, já que o tratamento de enlaces falhos é uma tarefa bem mais complicada. Sendo assim, ele não será destacado nas discussões e nos gráficos a seguir.

Tabela 3.1: Principais características dos algoritmos analisados

Características \ Algoritmo	Lee & Hayes 1992	J. Bruck 1994	Jie Wu 1994	Jie Wu 1995	Jie Wu 1996
Trata falhas de Enlace/Nó	**S/S**	S/N	S/N	N/S	S/N
Número máximo de falhas para garantir resultados desejados	$\lceil n/2 \rceil$	$\lceil n/2 \rceil - 1$	**n - 1**	**n-1**	**n - 2**
Número máximo de falhas para ainda ter a chance de obter resultados desejados	n.d.	n - 2	**$n2^{n-1} - 2^n - 1$**	n.d.	n.d.
Difusão a partir de qualquer nó	**S**	**S**	**S**	**S**	**S**
Tipo de informação de configuração de falhas	**informação local**	informação global	informação global	**informação local**	informação global limitada
Redundância	**N**	**N**	**N**	**N**	**N**
Estratégia	nó não-seguro	árvore binária	árvore binária / particionamento	níveis de segurança	árvore binária / um nó fonte por nível
Complexidade inicial (em *passos*)	$2n^3 + 5n^2 + 2n$	$2^n n.(3n - 3/4) + (5/2)(n^2 - n - 1)$	$4n^3 + 2n^2$	$n^3 + n^2 + 4n - 6$	**$n^2 + 5n$**
Complexidade apenas da difusão por etapa (em *passos*)	$n^2/2 + 8n + 5$	$2^{n-1} + 9$	**n + 12**	**4n + 6**	$n^3/8 + (21/8)n^2 + 2n + 5$
Número máximo de etapas para concluir a difusão	n + 1	**n**	n + 1	n + 1	**n**

Influência do número de difusões consecutivas

A Fig. 3.1 exibe o desempenho (em *passos*) dos algoritmos para um sistema de 64 nós quando o número de difusões consecutivas varia. Percebe-se que os algoritmos têm um comportamento quase linear sob estas circunstâncias. O algoritmo de Lee & Hayes (1992) apresenta o segundo melhor desempenho para até 33 difusões. O Algoritmo de J. Bruck (1994), que possui a curva com a inclinação mais suave, passa a apresentar o segundo melhor desempenho a partir de 33 difusões, apesar de já haver apresentado o pior deles (de 1 a 7 difusões). Este é talvez o melhor exemplo de como o número de difusões consecutivas pode influenciar o desempenho de um algoritmo.

Os algoritmos de Jie Wu de 1994 e 1996 têm uma participação modesta neste contexto, não sendo uma boa escolha em termos de desempenho para aplicações e/ou ambientes que exijam várias difusões.

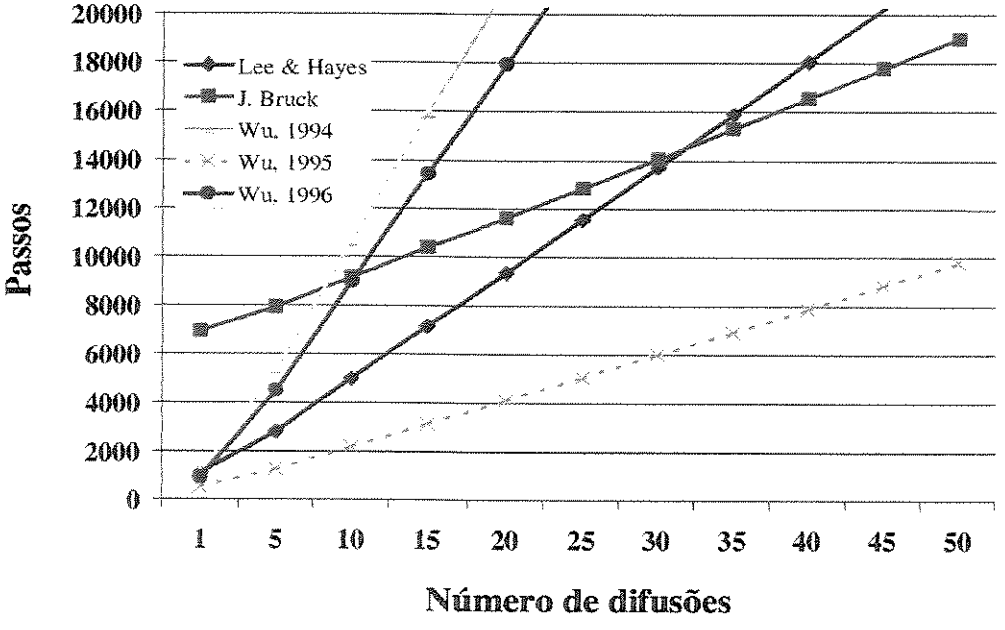

Figura 3.1: Comparação de desempenho com difusões consecutivas (n=6)

Desempenho para uma única difusão

A Fig. 3.2 apresenta o desempenho (em *passos*) dos algoritmos para uma única difusão quando o hipercubo varia de tamanho. Ao contrário do que se vê na Fig. 3.1, neste caso o algoritmo de J. Bruck (1994) apresenta o pior desempenho, especialmente para sistemas com mais que 8 nós. Conclui-se então que ele é bastante sensível ao crescimento do tamanho da rede e pouco reage ao aumento do número de difusões seqüenciais. Os algoritmos de Lee & Hayes (1992) e Jie Wu de 1994 e 1996 apresentam um comportamento similar. Entretanto, o algoritmo de Jie Wu de 1996 mostra uma pequena vantagem em relação aos outros dois. Sob estas circunstâncias, ele seria a segunda melhor opção quando só 1 difusão é necessária.

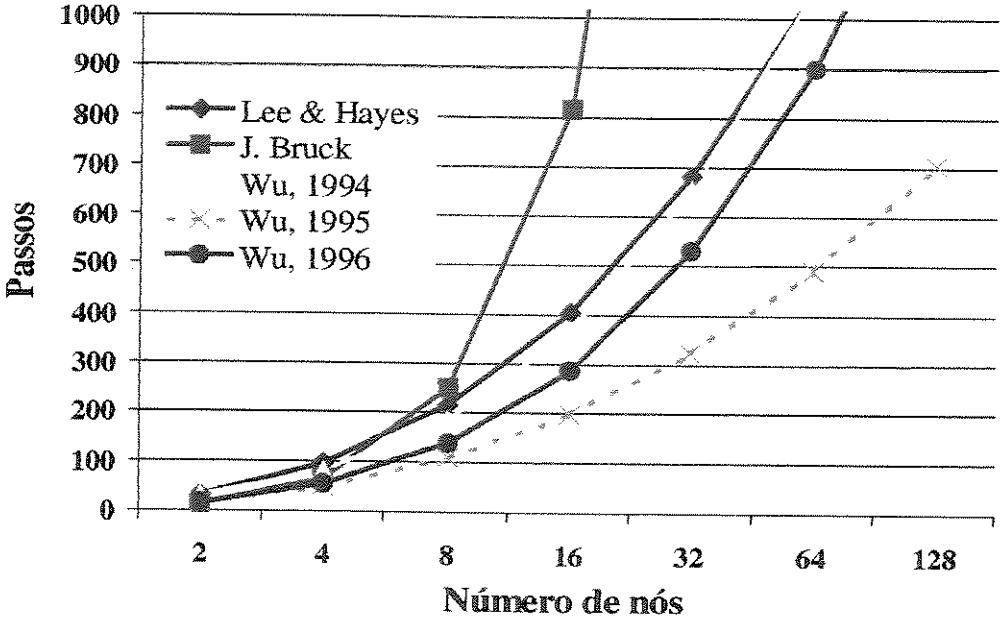

Figura 3.2: Comparação de desempenho para uma única difusão

Desempenho para 10 difusões consecutivas

A Fig. 3.3 exibe o desempenho (em *passos*) dos algoritmos para 10 difusões consecutivas quando o hipercubo cresce de 8 a 128 nós. As curvas para hipercubos com menos de 8 nós não apresentam qualquer informação relevante, por isso foram deixadas de fora do gráfico. Nota-se agora que o algoritmo de J. Bruck (1994) tem o segundo melhor desem-

penho para um sistema com até 32 nós. A partir de 32 nós o algoritmo de Lee & Hayes (1992) ganha esta posição. Este é um comportamento oposto comparado com o que se vê na Fig. 3.1. Um outro aspecto interessante e que vale ser ressaltado é a comparação entre o algoritmo de Jie Wu (1994) e o algoritmo de Lee & Hayes (1992). Para 32 nós, o desempenho do algoritmo de Jie Wu (6520 *passos*) é quase a metade do desempenho do algoritmo Lee & Hayes (3360 *passos*), enquanto que para 128 nós esta relação não é muito diferente ($\frac{16220}{7030} = 2,3$). Observa-se que apesar de ser, nestas condições, aproximadamente duas vezes mais lento que o seu concorrente, o algoritmo de Jie Wu (1994) tolera $n-1$ componentes falhos enquanto o algoritmo de Lee & Hayes tolera apenas $\lceil \frac{n}{2} \rceil$ componentes falhos.

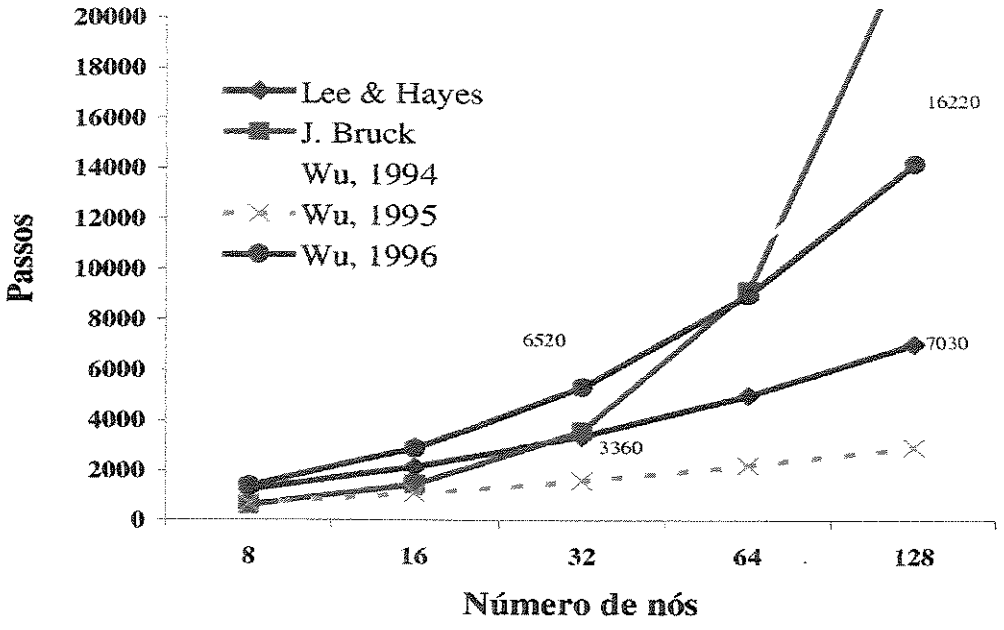

Figura 3.3: Comparação de desempenho para 10 difusões consecutivas

Desempenho para 100 difusões consecutivas

A Fig. 3.4 detalha o comportamento dos algoritmos para sistemas de médio a grande porte (de 64 a 4096 nós) e considerável número de difusões consecutivas (100). Considera-se nesta figura hipercubos de médio a grande porte por serem estes os casos onde se nota alguma alteração relevante nas curvas para este número de difusões consecutivas. Para até 64 nós o algoritmo de J. Bruck (1994) possui o segundo melhor desempenho; contudo, a partir de 128 nós, sua curva começa a crescer muito rapidamente e logo ele se torna uma

má opção em termos de desempenho. Para um sistema com mais de 128 nós, o algoritmo de Lee & Hayes é a segunda melhor opção quando o fator principal é o desempenho. Os algoritmos de Jie Wu (1995) e Lee & Hayes (1992) são os únicos que não apresentam uma estratégia de difusão baseada na análise da estrutura da árvore de difusão montada durante a operação. Ao invés de serem guiados por uma seqüência de dimensões, eles são guiados pelo nível de segurança da vizinhança dos nós. Esta é a razão pelo bom desempenho dos dois em relação aos outros, já que lidar com a estrutura da árvore binária de difusão demanda um esforço significativo em termos de tempo. Os algoritmos de Jie Wu de 1994 e 1996 têm um desempenho bem próximo um do outro, o que faz com que a escolha entre um e outro dependa de outros fatores além do desempenho.

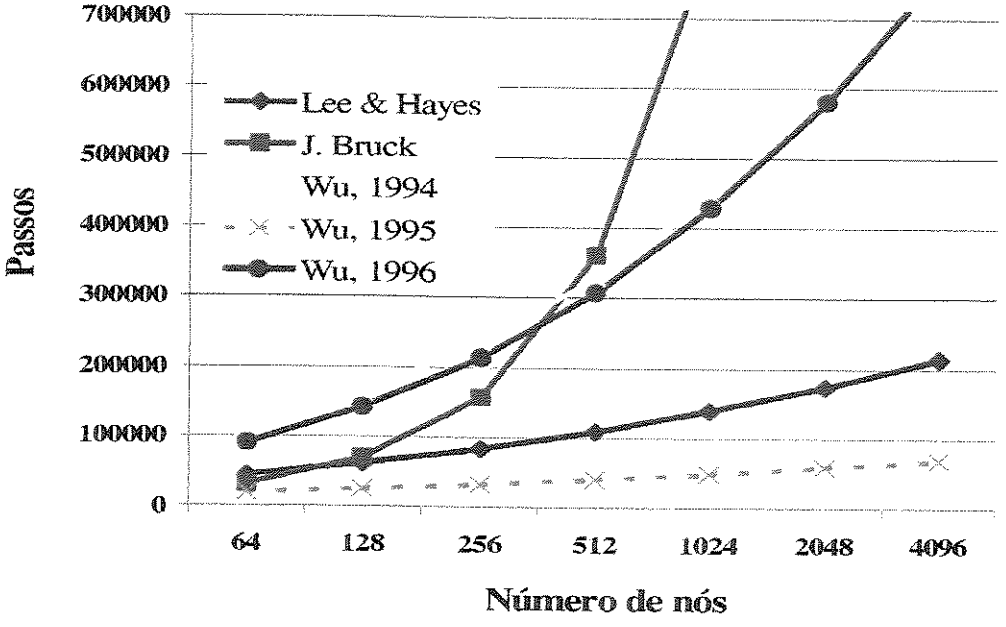

Figura 3.4: Comparação de desempenho para 100 difusões consecutivas

3.3 Conclusão

Este estudo apresentou as principais características dos algoritmos de difusão de mensagens tolerantes a falhas para hipercubos, destacando os aspectos que devem ser levados em conta durante a análise dos mesmos. Foram dados também exemplos de como, de posse

dos dados de alguns algoritmos, discernir a solução mais adequada para um determinado ambiente e/ou aplicação. Cinco algoritmos foram escolhidos, estudados e comparados, sendo que o resultado desta análise culminou com a construção de uma tabela e de gráficos que exibem as principais características de cada um, facilitando o processo de análise e comparação.

Os resultados desta pesquisa não somente facilitaram o processo de escolha para uma determinada situação entre os algoritmos existentes, mas também contribuíram para a elaboração de um novo algoritmo que apresentasse o que há de melhor nos algoritmos analisados. O conhecimento dos fatores mais importantes que definem um algoritmo de difusão como sendo eficiente e adequado à maioria das situações, somado às idéias que podem ser extraídas de cada solução e combinadas de forma a se complementarem, foram o ponto de partida para a proposição do algoritmo *ESLAB* apresentado no Cap. 5.

Mas, antes de apresentar este novo algoritmo é conveniente ressaltar que todos os que foram analisados, com exceção do de Jie Wu (1995), partem da suposição de que a configuração de falhas do sistema é previamente conhecida, sem mostrar como isto pode ser conseguido. No caso do algoritmo de Jie Wu (1995) a única informação sobre configuração de falhas que os nós necessitam é saber se seus vizinhos são ou não falhos, o que é facilmente conseguido através de trocas de mensagens de teste. No Cap. 4 este assunto é tratado com mais detalhes e nele é proposto um algoritmo simples e eficiente para a detecção e localização de falhas em hipercubos.

Capítulo 4

Detecção e Localização de Falhas

4.1 Introdução

O conceito de detecção e localização de falhas em sistemas multicomputadores é vasto e envolve vários aspectos que vão desde simples detalhes como queima de um componente físico até os mais complexos como comportamentos inesperados do sistema operacional, influência da oscilação de energia no sistema, etc. Devido a esta amplitude de conceito, este estudo deter-se-á apenas aos detalhes básicos e específicos do hipercubo.

Este trabalho não trata os casos de falhas transientes e intermitentes que são em geral mais freqüentes que as falhas permanentes [IYE 83] e podem estar relacionadas a falhas de software e de hardware. Em geral, as falhas que não são permanentes se apresentam como uma manifestação de uma falha a nível lógico (erro) e podem ser detectadas através de computadores que realizam auto-teste [REN 84] ou por uma série de testes funcionais que um computador realiza em outro [ARM 81] [KUH 81] [DIL 84]. As falhas permanentes, por outro lado, são normalmente defeitos físicos. Uma forma simples de tratar as falhas transientes, uma vez que se tenha conhecimento de que um nó específico vem apresentando tais falhas, é eliminar os processos que nele estão sendo executados, desativá-lo (torná-lo completamente inoperante) e executar um algoritmo de detecção de falhas permanentes para que ele seja detectado como sendo um nó com defeito físico. Desta forma, é possível combinar as duas abordagens de forma complementar.

Entre as formas de se contornar o problema de falhas em hipercubos, existem os sistemas que lançam mão de componentes reservas [REN 96], os que dividem o hipercubo

em subcubos menores e aproveitam aqueles livres de falhas (fault-free) [CHE 97], e os que simplesmente deixam tal tarefa para as aplicações [JWU'95] [JWU 95] [JWU 96] [BRU 94] [LEE 92] [LEU 96]. Cada uma delas apresenta vantagens e desvantagens [BAN 90]. Entretanto, acredita-se que as aplicações podem ser implementadas de maneira econômica, eficiente e tolerante a falhas, tirando proveito das peculiaridades do hipercubo.

4.2 Algoritmos existentes para detecção de falhas

Os métodos mais populares de detecção de falhas são em geral baseados em um dos três tipos de modelo:

- centralizado, que apresenta um agente monitor, que pode ser um nó do sistema ou um processador externo;
- hierárquico, com um conjunto de nós monitores formando uma estrutura de árvore onde as responsabilidades de monitoramento aumentam no sentido folha-raiz;
- distribuído, onde cada nó não-falho é capaz de diagnosticar o sistema, mesmo quando parte da rede se torna falha.

O modelo centralizado é inerentemente não-confiável, já que, se o nó monitor se torna falho, o gerenciamento é interrompido em todo o sistema. Similarmente, o modelo hierárquico falha no sentido de que se um dos nós monitores se torna falho o monitoramento cessa em parte do sistema. O modelo distribuído é o mais atraente e os trabalhos mais recentes sobre o assunto apresentam uma solução distribuída [RAN 95] [DUA 98] [MAI 99].

Entre os algoritmos mais atuais, pode-se destacar o proposto por E. Procópio, T. Nanya, G. Mansfield e S. Noguchi em [DUA 98]. Neste algoritmo, os nós testam seus enlaces em intervalos periódicos de tempo, sendo feito apenas um teste por enlace. Dos dois nós conectados por um enlace, aquele com o maior identificador (rótulo) é o responsável por testar o enlace. Os testes são esquematizados de tal forma que, em caso de falha de enlace ou de nó, os dois lados do enlace detectam a passagem de um intervalo de tempo pré-determinado (*time-out*) em que não há comunicação. Quando um novo evento (falha) é detectado, informação de diagnóstico é disseminada em paralelo para todos os nós da rede.

Essa solução garante que uma configuração de falhas denominada *jellyfish* seja sempre detectada. Esta configuração ocorre quando nós que testam uns aos outros e são responsáveis por retransmitir eventos de falha se tornam falhos ao mesmo tempo e os demais nós conectados não conseguem diagnosticar a situação.

Apesar de tratar enlaces falhos e apresentar uma série de vantagens, este algoritmo foi implementado visando redes de propósito geral e não redes de interconexão como o hipercubo, o que o impede de tirar proveito da estrutura regular do hipercubo. Além disso, embora seja dito que o tráfego de mensagens de teste no sistema não compromete o desempenho do mesmo, o fato de que a detecção de cada falha é disseminada a todos os nós do sistema constitui um desperdício de banda e um aumento desnecessário de custo de tempo para sistemas/aplicações que necessitam apenas de informação local ou global limitada para realizar difusões.

4.3 Algoritmo *DETECT*

4.3.1 Introdução

Como visto em capítulos anteriores, vários algoritmos já foram propostos para fazer com que operações comuns em hipercubos, como unicasting, multicasting, broadcasting e merging, se tornem tolerantes a um certo número de falhas. Os autores desses algoritmos, normalmente, partem do princípio de que os pontos de falha do hipercubo já são previamente conhecidos pelos nós operantes, porém nenhum deles especifica como isso pode ser feito e que algoritmo utilizar. Não se encontrou na literatura qualquer algoritmo específico para o hipercubo que pudesse fornecer as informações de configuração de falhas de forma distribuída e eficiente.

Neste capítulo é apresentada uma forma distribuída, simples e eficiente de, em um hipercubo binário, identificar (e difundir a informação sobre) elementos falhos (sejam nós, enlaces ou uma combinação destes) desde que seu número não exceda $n - 1$, onde n é a dimensão do hipercubo.

4.3.2 Arquitetura básica do algoritmo

O método de detecção de falhas em hipercubos que será apresentado baseia-se na troca de informações entre nós vizinhos em apenas três etapas distintas. As informações recolhidas pelos nós são armazenadas em um vetor de n posições e alfabeto 0,1,2 denominado *fault* [LEU 96]. Essa troca de informações entre nós é na realidade uma pequena mensagem que cada nó envia aos seus vizinhos para efeito de teste. Ela pode ser realizada logo que a máquina é ligada ou sempre que se deseja testar o estado atual de falhas do sistema.

A operação é iniciada com todos os nós enviando mensagens de teste aos seus vizinhos, configurando a primeira etapa de troca de mensagens. Em caso de assincronismo total entre os nós com relação ao início do processo, o algoritmo pode ser escrito de forma que um único nó inicie o processo e os demais sejam "despertados" assim que recebam a mensagem. A partir daí, a operação se desenvolverá sob um assincronismo parcial, controlado por *timeout*.

Inicialmente, todos os nós criam o vetor *fault* com todas as posições iguais a "1". Assim sendo, logo que um nó recebe uma mensagem, ele atualiza o vetor *fault* com um "0" na posição que indica a dimensão pela qual ele recebeu a mensagem. Após um certo período de tempo (*time-out*), caso não tenha recebido mensagem de um vizinho, ele assume a presença de falha, a qual pode ser de nó ou de enlace. Daí, o nó apresentará "1" em toda posição do vetor *fault* que corresponde a uma dimensão que apresenta falha.

Por exemplo, o nó 000 na Fig. 2.1 (Cap. 2) envia mensagem teste para os seus vizinhos 001, 010 e 100, porém só recebe resposta dos nós 010 e 100. Portanto seu vetor *fault* será [001], indicando que existe uma falha na dimensão 0. A princípio o nó 000 não tem como saber onde está a falha, se no nó 001, se no enlace 00$*$ ou em ambos. A forma de resolver tal problema será apresentada mais adiante.

Uma vez expirado o tempo (*time-out*), os nós iniciam a segunda etapa de troca de mensagens enviando seu próprio vetor *fault* a seus vizinhos. Cada nó recebe então até n vetores *fault* e os organiza como mostrado a seguir.

Cada vetor *fault* será inserido em uma linha de uma matriz F, linha esta que representa a dimensão pela qual ele recebeu o vetor *fault*. Todas as posições da matriz F são iniciadas com "1" e, à medida que os vetores *fault* vão chegando, cada linha da matriz é atualizada. Vejamos o caso do nó 100. Ele recebe do nó 000 o vetor [001], do nó 101 o vetor [010] e do nó 110 o vetor [001]. Ao final ele terá então a seguinte matriz.

$$\begin{array}{ll} \text{Dimensão } 0 \Rightarrow & \\ \text{Dimensão } 1 \Rightarrow & \\ \text{Dimensão } 2 \Rightarrow & \end{array} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{array}{l} \Rightarrow \text{recebida do nó } 101 \\ \Rightarrow \text{recebida do nó } 110 \\ \Rightarrow \text{recebida do nó } 000 \end{array}$$

Após receberem os vetores *fault* dos vizinhos, obviamente daqueles por dimensões não falhas, os nós realizam a operação descrita a seguir.

Para cada posição p_j igual a 1 no vetor *fault* vindo de cada dimensão d_i execute os seguintes passos:

- Obtenha o valor da posição p_i do vetor *fault* vindo da dimensão d_j.
- Se $p_i = 0$, então envie ao nó vizinho pela dimensão d_i uma mensagem informando que ele possui um enlace falho na dimensão p_j.
- Se $p_i = 1$, não há informação suficiente para determinar a localização da falha, e o algoritmo se repete para outros valores de i e j.

A idéia é que, se existe um nó x que sabe que um nó y não consegue se comunicar com um nó z, mas existe um nó w que consegue, então o nó x tem conhecimento de que z não é falho e sim o enlace entre ele e nó y é que é falho. Para o caso do exemplo acima, temos que todos os vetores *fault* apresentam valor 1. No caso daquele vindo de d_2, a posição que apresenta valor 1 será então p_0. O valor da posição p_2 para o vetor da dimensão d_0 é zero. Assim sendo o nó 100 enviará mensagem ao nó 000 informando que ele apresenta um enlace falho na dimensão 0. O envio da notificação aos nós que apresentam enlaces falhos constitui a terceira etapa da troca de mensagens do método. No caso dos vetores vindos das dimensões d_0 e d_1, as respectivas posições que apresentam valor 1 são p_1 e p_0; logo, o nó 100 nada consegue concluir a respeito das falhas destes vizinhos. Note que um nó que apresenta enlace falho pode vir a receber esta notificação de mais de um nó vizinho.

Numa aplicação onde não somente os nós necessitam saber se possuem ou não enlaces falhos mas também seus vizinhos necessitam saber desta informação, é necessária uma quarta troca de mensagens para que os nós que possuem vizinhos não-perfeitos atualizem a matriz F. Assim, quando um nó recebe comunicação de um vizinho informando que este último é não-perfeito, o nó atualiza seu vetor *fault* colocando o valor "2" na posição referente àquele vizinho. Desta forma, o valor "2" numa posição do vetor *fault* indica a presença de

um nó não-perfeito pela dimensão associada àquela posição.

O algoritmo detalhado é apresentado no Apêndice B.1.

Demonstra-se a seguir que o Algoritmo *DETECT* é correto no sentido de que sempre é capaz de detectar e localizar falhas em um n-cubo quando estas não ultrapassarem o número de $n-1$.

LEMA 1: Cada enlace em um n-cubo pertence a $(n-1)$ subcubos de ordem dois (2-cubos) distintos.

DEMONSTRAÇÃO:

Assim como visto no Cap. 2, os 2-cubos (subcubos de ordem 2) contidos em um hipercubo de maior dimensão podem ser representados por rótulos de n bits e alfabeto 0,1,$*$. Eles são da forma $[0,1]^i * [0,1]^j * [0,1]^k$, onde $[0,1]^m$ é uma seqüência de m bits ($m \geq 0$). Por exemplo, na Fig. 4.1 temos o 2-cubo $*0*$ formado pelos nós 001, 000, 101 e 100, e pelos enlaces $*01$, $10*$, $00*$ e $*00$. (Nota: apesar de seqüências de bits com um asterisco poderem ser interpretadas como sendo um 1-cubo, neste trabalho elas significam o enlace entre os dois nós deste 1-cubo.)

O número de 2-cubos presentes em um hipercubo de n dimensões ($n \geq 2$) pode ser facilmente calculado a partir de sua representação. Seu rótulo apresenta sempre dois $*$'s, que podem estar em qualquer posição do mesmo em um total de combinações de n dois a dois, e $n-2$ outros bits que podem ter valor "1"ou "0". Portanto, o número Z de 2-cubos em um n-cubo pode ser calculado pela Eq.(4.1).

$$Z = \binom{n}{2} \cdot 2^{n-2} = \frac{n!}{(n-2)!2!} \cdot 2^{n-2} = \frac{n(n-1)}{2} \cdot 2^{n-2} = \frac{(n-1)}{4} \cdot n2^{n-1} \qquad (4.1)$$

Cada um dos Z 2-cubos possui 4 enlaces. Logo, multiplicando ambos os lados da Eq.(4.1) por 4 teremos um total de L_2 enlaces existentes nos Z 2-cubos, incluindo repetições.

$$4Z = (n-1) \cdot n2^{n-1} = L_2 \qquad (4.2)$$

Como o número de enlaces em um n-cubo é $L_n = n2^{n-1}$, então pode-se concluir que $L_2 = (n-1)L_n$, ou seja, o número total de enlaces necessários para formar Z 2-cubos é

$n-1$ vezes maior que o número total de enlaces no sistema, o que indica que cada enlace deve estar presente em exatamente $(n-1)$ 2-cubos, já que o hipercubo é uma estrutura regular. O exemplo da Fig. 4.1 mostra que o enlace $*00$ no 3-cubo pertence a apenas dois 2-cubos, o $(*0*)$ e o $(**0)$ indicados pelas linhas pontilhadas. $\triangledown$

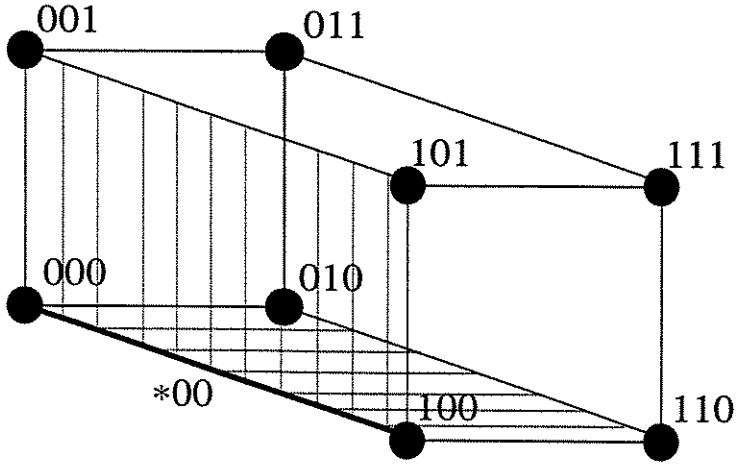

Figura 4.1: Enlace compartilhado por dois 2-cubos

LEMA 2: Dois enlaces, ou um enlace e um nó que não estejam conectados, tomam parte juntos em no máximo um subcubo de ordem dois.

DEMONSTRAÇÃO:

Na representação de um 2-cubo os asteriscos ocupam dimensões específicas no rótulo. Um enlace l_1 qualquer cujo asterisco está na dimensão d estará presente em um 2-cubo que apresentar um de seus asteriscos na dimensão d e for equivalente em 0's e 1's em todos os outros bits do rótulo, exceto onde se encontra o outro asterisco. Isto é fácil de perceber pois o rótulo do 2-cubo deve abranger o rótulo do enlace. Assim, por exemplo, o enlace $001*0$ faz parte do 2-cubo $*01*0$, já que este abrange os enlaces $001*0$ e $101*0$.

Supondo agora que os enlaces l_1 e l_2 estão presentes em um mesmo 2-cubo k_1 que apresenta os asteriscos nas dimensões d_1 e d_2. Mostra-se que não existe nenhum outro 2-cubo que contenha l_1 e l_2. Sabendo que um outro 2-cubo qualquer k_2 não pode ter a mesma representação de k_1 (k_2 diferirá de k_1 na posição dos asteriscos e/ou em pelo menos

um bit nas outras posições do rótulo) tem-se que:

1. se l_1 e l_2 têm seus asteriscos na mesma posição, por exemplo d_1, tem-se que:
   (a) se k_2 apresenta um asterisco em d_1, então:
      i. se o outro asterisco de k_2 está em d_2, então pelo menos um dos seus outros bits difere de k_1; logo este bit também diferirá do seu correspondente em l_1 e l_2, e k_2 não poderá conter estes enlaces.
      ii. se o outro asterisco de k_2 não está em d_2, então ele terá 0 ou 1 naquela posição, o que permite que k_2 contenha l_1 ou l_2, mas não os dois, pois l_1 e l_2 possuem bits diferentes naquela posição.
   (b) se k_2 não apresenta um asterisco em d_1, então l_1 e l_2 não poderão estar contidos em k_2 já que as posições dos asteriscos não coincidem.
2. se l_1 e l_2 têm seus asteriscos em posições diferentes, tem-se que:
   (a) se k_2 apresenta os asteriscos em d_1 e d_2, então pelo menos um dos seus outros bits difere de k_1; logo este bit também diferirá do seu correspondente em l_1 e l_2, e k_2 não poderá conter nenhum dos dois enlaces.
   (b) se um asterisco de k_2 coincide com um dos asteriscos de k_1, d_1 por exemplo, então os dois enlaces não poderão tomar parte juntos em k_2, já que somente aquele cujo asterisco esteja em d_1 poderá tomar parte neste subcubo.
   (c) se k_2 não tem asteriscos em d_1 nem d_2, então nem l_1 nem l_2 poderão estar presentes neste subcubo, já que não há coincidência dos asteriscos dos enlaces com os do subcubo.

Analisa-se agora o caso da presença de um enlace l e um nó v não conectado a l em um mesmo 2-cubo k_1 cujos asteriscos estão nas dimensões d_1 e d_2. Mostra-se que não existe nenhum outro 2-cubo que contenha l e v. Considere-se, sem perda de generalidade, que l possui o asterisco na dimensão d_1 e um 1 na dimensão d_2. Como um 2-cubo abrange 4 nós, dois deles apresentam 1 na dimensão d_2 (os conectados por l) e os outros dois, entre eles v, 0 na dimensão d_2. Como já visto, caso um 2-cubo k_2 tenha os asteriscos nas dimensões d_1 e d_2, deverá obrigatoriamente ter pelo menos um bit diferente de k_1 nas outras posições do rótulo, o que faz com ele não contenha nem l nem v. Caso k_2 não apresente um asterisco em d_1, também não conterá l. Caso k_2 não apresente um asterisco em d_2, esta dimensão

conterá 1 ou 0. Se for 0, k_2 não conterá l, já que l possui um bit 1 na posição d_2. Se for 1, k_2 não conterá v, pois v possui um bit 0 na posição d_2. Assim sendo, a presença de l e v em outro 2-cubo qualquer torna-se impossível.

Uma outra maneira de se demonstrar o LEMA 2, de uma forma mais informal e fácil de se entender, é apresentada a seguir. A forma como os elementos de um hipercubo estão representados baseia-se num sistema de coordenadas n-dimensional. Assim, é possível aplicar a eles o conceito de plano. Sabe-se da geometria analítica que duas retas podem formar um único plano. Como na representação do hipercubo os enlaces podem ser vistos como segmentos de retas distintas e os 2-cubos (quadrados) podem ser vistos como partes de planos distintos, cada par de enlaces pode formar no máximo um único plano, ou seja, fazer parte de um único 2-cubo.

De forma similar, sabe-se também que uma reta e um ponto que não esteja sobre ela formam também um único plano. Aplicando o conceito de ponto aos nós de um hipercubo, verifica-se que um enlace e um nó que não esteja conectado a este enlace participam em no máximo um plano, ou seja, um único 2-cubo. $\bigtriangledown$

TEOREMA 1: Na presença de um número de falhas inferior a n em um n-cubo, o algoritmo *DETECT* consegue detectar e localizar todas elas.

DEMONSTRAÇÃO:

Para a demonstração, analisa-se primeiramente o caso do 2-cubo e em seguida generaliza-se para o caso de um n-cubo.

A Fig. 4.2(a) apresenta um 2-cubo com um enlace falho entre os nós 00 e 10. Vale ressaltar que o 2-cubo pode apresentar no máximo 1 falha de acordo com as considerações feitas aqui.

Nesta figura é fácil perceber que o nó 11, tendo conhecimento dos vetores de falhas dos vizinhos 01 e 10, consegue avaliar a condição do enlace $*0$. O mesmo pode-se dizer do nó 01 quando analisa os vetores de falhas de seus vizinhos 00 e 11. Após conferir os vetores fault de seus vizinhos, o nó 11 analisará a situação da seguinte forma: se meu vizinho na dimensão 0 (nó 10) não se comunica com seu vizinho na dimensão 1 (nó 00) e o meu vizinho na dimensão 1 (nó 01) consegue se comunicar com seu vizinho na dimensão 0 (nó 00), então 00 não é falho e sim o enlace que o conecta ao meu vizinho na dimensão 0 é que é falho.

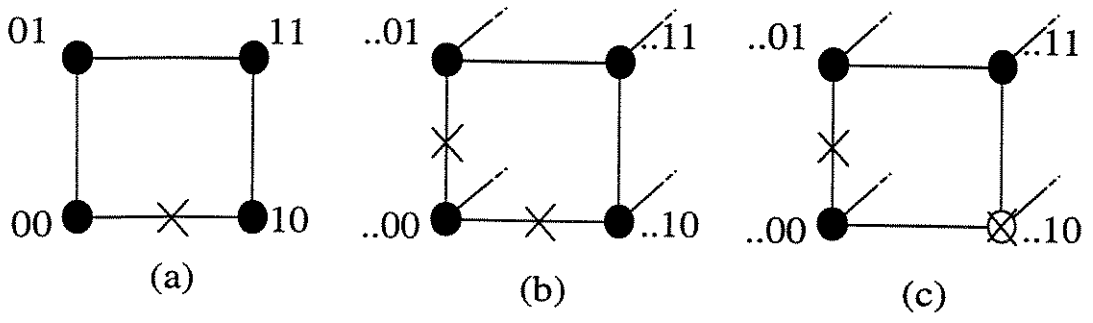

Figura 4.2: Exemplos de falhas em um 2-cubo

Pode acontecer, entretanto, a situação mostrada na Fig. 4.2(b) para o caso de um hipercubo de dimensão maior que 2. Para o n-cubo mostrado nesta figura percebe-se que o algoritmo *DETECT* não é capaz de fazer com que $[0,1]^i11$ e $[0,1]^i01$ percebam que $[0,1]^i*0$ é falho nem que $[0,1]^i10$ e $[0,1]^i11$ percebam que $[0,1]^i0*$ é falho. Porém, como o objetivo é fazer com que todos os nós conectados por um enlace falho sejam notificados disso, basta garantir que pelo menos um dos vizinhos desses nós falhos faça tal notificação.

A idéia do algoritmo *DETECT* é analisar todos os 2-cubos do hipercubo que apresentam algum enlace falho. Se para cada enlace falho existir pelo menos um 2-cubo do qual ele faz parte e que não apresente nenhuma outra falha, assim como exemplificado no caso da Fig. 4.2(a), os nós que ele conecta sempre serão notificados da falha. Mostra-se que tal 2-cubo sempre existirá para o caso de um n-cubo com menos de n falhas.

Seja um enlace l falho. Ele pertence a $(n-1)$ 2-cubos distintos (LEMA 1). Como o número máximo de elementos falhos no sistema não pode exceder $n-1$, somente mais $n-2$ elementos falhos são permitidos além de l. No pior caso cada um destes $(n-2)$ elementos falhos formará um, e somente um (LEMA 2), 2-cubo com o enlace l, definindo um número máximo de $(n-2)$ 2-cubos distintos similares aos da Fig. 4.2(b) ou (c). Portanto, o enlace l estará presente em pelo menos $(n-1)-(n-2)=1$ subcubo como o da Fig. 4.2(a), o que viabiliza a detecção e localização deste enlace falho pelo Algoritmo DETECT. $\bigtriangledown$

4.3.3 Análise de desempenho

Abaixo é detalhado o esforço computacional local (em cada nó) para cada etapa do algoritmo *DETECT*. A ordem do custo de cada etapa pode ser avaliada em cada passo executado.

Etapa	Custo
Inicialização de variáveis	$O(n)$
Envio de mensagem teste a cada vizinho	$O(n)$
Recebimento e armazenamento das respostas	$O(n)$
Envio do vetor *fault* a cada vizinho	$O(n)$
Conferência, armazenamento e atualização de variáveis para cada resposta recebida	$O(n)$
Análise de cada vetor *fault* e informação aos nós vizinhos em caso de detecção de enlace falho	$O(n^2)$
Conferência, armazenamento e atualização de variáveis para cada notificação recebida	$O(n)$

Custo global = $O(n^2)$ passos.

Uma análise geral do código apresentado no Apêndice B.1 revela um custo total de operação, em passos (sendo a definição de passo a mesma adotada no Cap. 3), descrito pela Eq.(4.3).

$$\mathbf{T_n = 2n^2 + 11n(passos)}. \tag{4.3}$$

O algoritmo *DETECT* foi implementado em C e testado extensivamente em um hipercubo n-Cube2, de 64 processadores com 4 Mbytes de memória cada um, da empresa NCUBE [NCU 90]. Foram realizados testes para subcubos de 3, 4, 5 e 6 dimensões. Os casos (configurações) de falhas possíveis, para até $n-1$ falhas e para cada um desses subcubos é apresentado no Apêndice A.3. Foram feitos testes para todas as configurações de falhas para os hipercubos de 3 e 4 dimensões, que foram, respectivamente, de 170 e 14702 configurações. Devido à explosão combinatória de casos de falhas para hipercubos de dimensões maiores e o tempo gasto com os testes, seria inviável a execução de todos eles. Assim sendo, decidiu-se testar para o hipercubo de 5 dimensões, que possui o dobro do número de nós do hipercubo de 4 dimensões, exatamente o dobro de casos de falhas do hipercubo de 4 dimensões, ou seja, 29404 casos. A mesma regra foi usada para o hipercubo de 6 dimensões tendo-se testado 58808 casos de falhas. Tanto os casos de falhas para o hipercubo de 5 dimensões quanto para o hipercubo de 6 dimensões foram gerados aleatoriamente. Essas falhas foram geradas de forma a garantir sua singularidade, isto é, de forma a não haver repetições de casos. Esses testes não foram realizados simplesmente para efeito de análise de desempenho, mas também para confirmar que o algoritmo *DETECT*

consegue detectar e localizar todas as falhas corretamente. As falhas no hipercubo foram implementadas por software da seguinte maneira. Um vetor contendo os identificadores de todas as falhas é passado como parâmetro para um programa que executa o algoritmo *DETECT*, de forma que os nós cujos identificadores estão presentes neste vetor abortam o processo, e os nós que possuem enlaces cujos identificadores estão presentes neste vetor não enviam mensagens por estes enlaces.

Os valores de tempo registrados durante as simulações (ver Apêndice A.2 para maiores detalhes sobre a medição dos tempos) são apresentados na TABELA 4.1. A coluna de tempo médio é referente à média dos valores registrados durante os testes, em função de n, e o desvio padrão e o erro padrão são relativos a esta média. A TABELA 4.2 apresenta o desempenho do algoritmo quando não existem falhas no sistema. É interessante notar que o algoritmo *DETECT* tem desempenho sensível ao número de falhas no sistema, o que já era de se esperar. Esta é uma característica positiva já que o algoritmo só demanda mais tempo quando é necessário, sendo o mais eficiente possível na ausência de falhas.

Tabela 4.1: Tempo para detectar $n-1$ falhas

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	6248	709	54
4	9407	1062	9
5	11181	1775	10
6	13526	2693	11

Tabela 4.2: Tempo de execução em sistema sem falhas

N^o de dimensões	Tempo (μs)
3	4895
4	6551
5	8225
6	9924

Após alguns testes de desempenho realizados no n-Cube2, descobriu-se que há uma diferença de tempo de processamento significativa (até 200 vezes) entre as chamadas de sistema para envio e recebimento de mensagens e as demais operações realizadas (ver Apêndice A.1). Este fato deve-se à forma como estão implementadas as chamadas de tais funções e a

interação entre o sistema de roteamento e o sistema operacional do n-Cube2. Em um sistema que possibilitasse uma otimização dos processos de envio e recebimento de mensagens, tal discrepância não existiria. A Eq.(4.4), que rege o desempenho do algoritmo utilizado nos testes no n-Cube2, foi proposta considerando-se valores de tempo de cada passo obtidos em testes independentes.

$$\mathbf{T_n = 195n^2 + 1453n + 8(\mu s)}. \tag{4.4}$$

É evidente que o tempo gasto em cada passo varia de acordo com a implementação, mas o mais importante é confirmar que a complexidade do algoritmo é $O(n^2)$. Vale observar que todos os nós executam o Algoritmo *DETECT* simultaneamente e que a equação acima foi obtida do código implementado em C. A Fig. 4.3 apresenta um gráfico comparativo entre as curvas com os valores médios de tempo medidos durante os testes e a curva da Eq.(4.4). Nota-se que há uma discrepância considerável entre as duas curvas para o resultado das simulações para o hipercubo de 6 dimensões. Essa diferença provém provavelmente do fato da equação retratar o pior caso. Percebe-se, por exemplo, que a soma do valor médio com o desvio padrão para 6 dimensões aproxima-se bastante do valor indicado pela equação. O intervalo de segurança (definido como duas vezes o valor do erro padrão) não foi inserido no gráfico por ser muito pequeno.

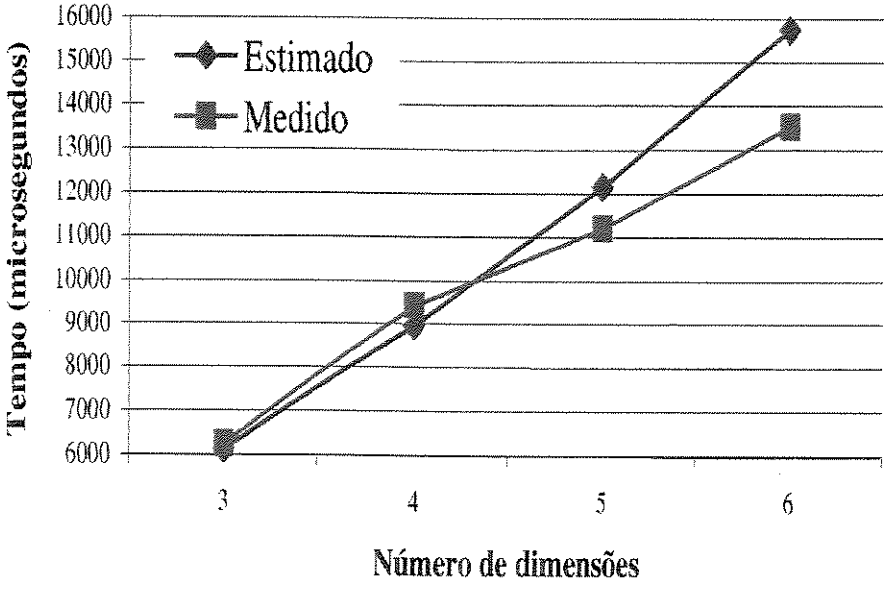

Figura 4.3: Desempenho de *DETECT*

4.3.4 Comparação com outros algoritmos

O algoritmo *DETECT* foi especificamente desenvolvido para prover informações sobre falhas em hipercubos a outros algoritmos que venham a necessitar desse tipo de informação, como o caso dos algoritmos de difusão de mensagens. Portanto, ele consegue tirar proveito das características mais importantes do hipercubo para realizar a detecção e localização das falhas de forma mais eficiente. Ao que parece, nenhum outro algoritmo tão específico quanto o *DETECT* foi desenvolvido até agora. O algoritmo proposto em [DUA 98], como comentado na Seção 4.2, apesar de tratar o caso dos enlaces falhos e ser distribuído assim como o *DETECT*, não é adequado para efeito de comparação pelo fato de tratar falhas transientes, o que o torna mais complexo que o *DETECT*, e ter de permanecer executando sem interrupção nos nós, alocando recursos que podem ser escassos caso os nós precisem executar aplicações ou outras operações. Se utilizado para realizar a mesma função para a qual o *DETECT* foi desenvolvido, ele certamente mostraria um desempenho inferior.

4.4 Conclusão

Algoritmos que provêem tolerância a falhas em hipercubos geralmente assumem que a informação sobre os elementos falhos já estão disponíveis de alguma forma para os nós e não detalham este aspecto do problema. Neste capítulo propõe-se um algoritmo distribuído para detecção e localização de falhas em hipercubos baseado em trocas de mensagens. O mesmo consegue detectar e localizar todas as falhas de enlace ou nó desde que, em conjunto, elas não ultrapassem o valor $n-1$, sendo n o número de dimensões do hipercubo. A complexidade do algoritmo é $O(n^2)$. Experimentos foram feitos e comprovaram que o algoritmo é não somente correto mas também apresenta desempenho satisfatório e compatível com o número de falhas no sistema.

Um ponto que necessita ser mais trabalhado no algoritmo *DETECT* está relacionado com a questão do intervalo de tempo (*timeout*) que os nós esperam tanto na primeira etapa do processo, quando os nós que não recebem resposta de um vizinho passam a considerá-lo falho, quanto na etapa final em que os nós esperam receber alguma notificação de algum vizinho a respeito de possuírem ou não enlaces falhos. Deve-se ser bastante cuidadoso ao se definir este intervalo de tempo pois, se ele for muito curto, um nó pode assumir uma situação falsa baseado no fato de não ter recebido uma mensagem que ainda está a caminho, e se ele for muito longo isto terá um impacto negativo no desempenho do algoritmo.

Capítulo 5

Algoritmo *ESLAB*

Foi apresentado no Cap. 3 uma análise e comparação de algoritmos desenvolvidos para realizar difusão de mensagens em hipercubos defeituosos. Como visto, diversas são as propriedades que caracterizam estes algoritmos. Cada uma das soluções estudadas possui um destaque, algo que a torna mais vantajosa que as outras em certo(s) aspecto(s). Porém, nota-se a necessidade de uma solução que combine os pontos positivos de cada um desses algoritmos, apresentando uma estratégia que supere todos os demais na maioria dos aspectos e condições. Um algoritmo com estas características foi desenvolvido e será apresentado em detalhes neste capítulo com a denominação de *ESLAB* (*Extended Safety Level Algorithm for Broadcasting*).

5.1 O conceito dos níveis de segurança

Uma das estratégias utilizadas pelo algoritmo *ESLAB* é a dos níveis de segurança que foi introduzida por Jie Wu em [JWU 95]. Ela foi escolhida por, entre outros aspectos, necessitar apenas de informação local no que se refere a falhas do sistema. Como pode ser visto no Cap. 3, acredita-se ser esta uma das razões pelo sucesso do algoritmo Jie Wu 1995 nas comparações de desempenho. Uma outra característica importante desta estratégia é o fato dela tolerar com segurança até $n-1$ falhas para um hipercubo de n dimensões. Mais falhas podem ser toleradas dependendo da distribuição das mesmas pelo sistema. O valor $n-1$ é o número máximo de falhas que podem existir em um sistema sem a ocorrência de

uma disjunção no hipercubo, já que n falhas já são suficientes para isolar um nó do resto do sistema. A questão dos hipercubos disjuntos não é tratada neste trabalho.

5.1.1 Definição dos níveis de segurança

Para cada nó não-falho em um hipercubo n-dimensional atribui-se um nível de segurança, que é um número, entre 0 e n inclusive, que dá uma medida aproximada da configuração de falhas da vizinhança de um nó. Considere $S(a) = k$ como sendo o nível de segurança do nó a. O nó a é então dito ser de segurança k, ou k-seguro. Nós falhos são rotulados como 0-seguros (nível de segurança mais baixo) enquanto os nós n-seguros (também chamados simplesmente de seguros) são aqueles que apresentam o nível mais alto de segurança. Um nó é chamado de não-seguro se seu nível de segurança for inferior a n. Descreve-se abaixo a forma de se calcular o nível de segurança de um nó.

Definição: Seja $(S_0, S_1, S_2, ..., S_{n-1})$, $0 \leq S_i \leq n$, a seqüência de níveis de segurança, em ordem crescente, dos n vizinhos do nó a no n-cubo, tal que $S_i \leq S_{i+1}$, $0 \leq i \leq n-1$. O nível de segurança do nó a é definido como n se $(S_0, S_1, S_2, ..., S_{n-1}) \geq (0, 1, 2, ..., n-1)$, onde $seq_1 \geq seq_2$ se, e somente se, cada elemento em seq_1 for maior ou igual ao seu correspondente em seq_2. Caso contrário, ele é definido como sendo k, onde $(S_0, S_1, S_2, ..., S_{k-1}) \geq (0, 1, 2, ..., k-1)$, $(S_k = k-1)$ e $k < n$.

Basicamente, a definição acima diz que após ordenados os níveis de segurança dos vizinhos de um nó em um vetor, seguindo do nível mais baixo para o mais alto, se algum desses níveis for menor que a posição que ele ocupa no vetor, então o nível de segurança do nó será 1 a mais que aquele nível. Por exemplo, se após receber os níveis de segurança dos vizinhos e ordena-los, um nó apresentar um vetor como o mostrado abaixo, seu nível de segurança será 1+1 = 2, já que $V(2) = 1$ e $V(2) < 2$.

i	0	1	2	3	4
V(i)	1	1	1	4	5

Para que os nós cheguem ao resultado definitivo de seus níveis de segurança, são necessários $n-1$ estágios de troca de informações entre vizinhos, sendo que no primeiro, todos os nós não-falhos consideram a si próprios como sendo seguros (nível de segurança n).

5.1.2 Difusão de mensagens utilizando os níveis de segurança

Uma vez que os nós terminam o cálculo dos seus níveis de segurança definitivos, eles compartilham estes dados com seus vizinhos e os armazenam em memória. Assim, eles estão aptos a iniciar um processo de difusão de mensagens baseado nos níveis de segurança. Os níveis de segurança só precisam ser recalculados caso se deseje detectar o aparecimento de uma nova falha no sistema.

Um nó k-seguro pode ser visto como um nó capaz de atingir todos os nós não-falhos em até k dimensões de distância dele durante um processo de difusão de mensagem. Assim sendo, somente um nó seguro (n-seguro) pode iniciar uma difusão com garantia de sucesso, ou seja, atingir todos os nós não-falhos do hipercubo. Se um nó não-seguro deseja iniciar uma difusão, ele primeiro deve enviar a mensagem a um nó vizinho seguro, cuja existência é garantida. Então, este vizinho seguro procede como se ele fosse o nó fonte original. Neste caso, a mensagem não será reenviada ao nó fonte original.

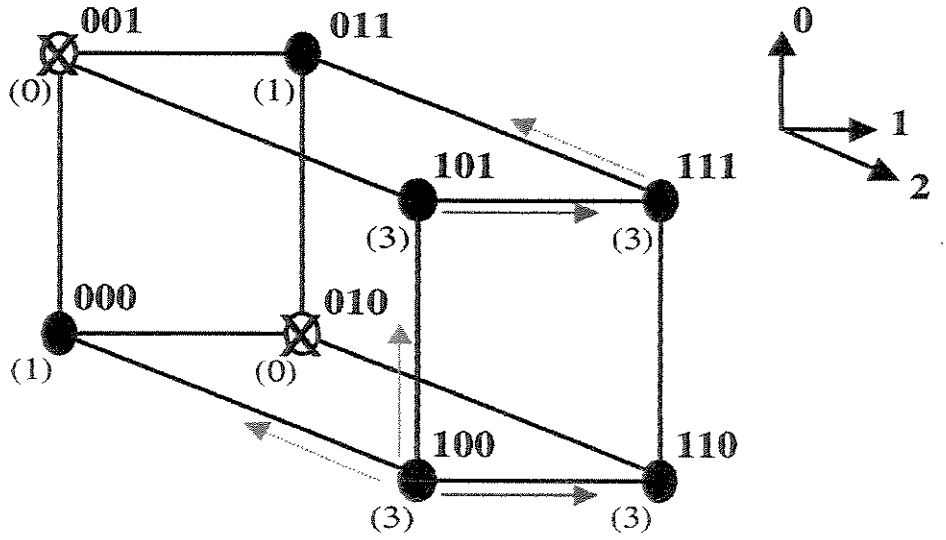

Figura 5.1: Difusão utilizando níveis de segurança

O processo de difusão baseado nos níveis de segurança consiste em dividir o cubo ao meio recursivamente, designando sempre um nó para iniciar a difusão em cada metade. Os nós são escolhidos de acordo com seus níveis de segurança. Por exemplo, no processo de difusão em um hipercubo de 3 dimensões representado na Fig. 5.1, o nó 100 é o nó origem.

Seu nível de segurança é $n = 3$ (como mostrado logo abaixo do nó). Logo ele é um nó seguro e pode iniciar uma difusão com garantia de sucesso. Seus vizinhos nas dimensões 0, 1 e 2 possuem níveis de segurança 3, 3 e 1, respectivamente. Logo, na primeira etapa da difusão, ele escolhe o vizinho pela dimensão 0 (101) para difundir a mensagem[1]. Dessa forma, o nó 100 designa o nó 101 como o responsável por levar adiante a difusão no subcubo $**1$. Na segunda etapa da difusão o nó 100 escolhe seu vizinho pela dimensão 1 para ser o próximo a receber a mensagem a partir dele, já que o nó 110 possui nível de segurança maior que o do nó 000. Assim, o nó 110 passa a ser o responsável pela difusão no subcubo $*10$. O nó 101, por sua vez, seguindo a mesma regra, escolhe a dimensão 1 para difundir a mensagem, designando o nó 111 como o responsável por difundir a mensagem no subcubo $*11$. Na terceira e última etapa o nó 100 envia a mensagem ao nó 000 e o nó 111 envia a mensagem ao nó 011. Os nós 000 e 011, que são os nós não-falhos com menor nível de segurança, são então folhas na árvore de difusão. São necessárias n etapas para finalizar uma difusão em um n-cubo quando o nó origem é seguro e $n + 1$ quando o nó origem é não-seguro.

No artigo [JWU 95] mostra-se que o uso dos níveis de segurança garante uma difusão com sucesso se o número de nós falhos no sistema for menor que n. Entretanto, a difusão pode falhar caso haja ocorrências de enlaces falhos. Isto pode ser facilmente percebido pelo fato de que, de acordo com a definição dos níveis de segurança, um nó pode ser seguro e ainda assim apresentar um enlace falho; e sendo seguro ele está apto a iniciar uma difusão. Caso ele inicie uma difusão, ele será responsável por enviar a mensagem a todos os seus vizinhos não-falhos, falhando então em enviar a mensagem ao vizinho não-falho conectado a ele pelo enlace falho.

5.2 O conceito estendido dos níveis de segurança

Nesta seção, será definido o conceito estendido de nível de segurança utilizado pelo algoritmo *ESLAB*. Também serão apresentados três lemas que sustentam a base do teórica do algoritmo.

O conceito dos níveis de segurança da forma como apresentado anteriormente, limita os algoritmos de difusão que o queiram utilizar já que poderão tratar apenas o caso de nós falhos no sistema. Esta é uma limitação muito forte, pois as ocorrências de enlaces falhos

[1] Poderia igualmente ter difundido para 110 que possui o mesmo nível de segurança de 101.

costumam ser mais freqüentes que as de nós falhos. Assim sendo, para que o algoritmo *ESLAB* possa utilizar as vantagens oferecidas pelo conceito dos níveis de segurança e ainda tratar o caso de enlaces falhos, é necessário estender este conceito. Esta extensão, batizada de conceito estendido dos níveis de segurança é a seguinte: **cada nó que apresenta enlace(s) falho(s) enxerga o(s) nó(s) na outra extremidade deste(s) enlace(s) como falho(s), isto é, como sendo 0-seguro(s).** O resto da definição original dos níveis de segurança permanece inalterado.

O conceito estendido dos níveis de segurança não garante que um processo de difusão de mensagem terá sucesso caso apenas ele seja usado como estratégia de difusão. Por isso, o algoritmo *ESLAB* lança mão de uma ferramenta extra a ser utilizada em conjunto com o conceito estendido dos níveis de segurança para que, em um hipercubo n-dimensional com até $n-1$ falhas, sejam elas de enlaces, de nós ou uma combinação de ambos, se consiga sempre realizar uma difusão com sucesso a partir de qualquer nó não-falho.

Antes de discutir os aspectos desta ferramenta, serão apresentados três lemas que formalizam as principais características do conceito estendido dos níveis de segurança, ajudam no entendimento do funcionamento do algoritmo e dão suporte à comprovação da validade do mesmo.

LEMA 1: Os nós não-falhos que não são atingidos durante uma difusão baseada apenas no conceito estendido dos níveis de segurança são nós não-perfeitos.

DEMONSTRAÇÃO:

A Fig. 5.2 apresenta uma árvore de difusão genérica para um hipercubo de n dimensões. Os níveis de segurança dos nós apresentados nesta árvore são os mínimos possíveis (o que significa que estes níveis poderiam ser maiores mas nunca menores) considerando-se que a difusão foi realizada obedecendo-se o conceito estendido dos níveis de segurança e que eles estão ordenados de forma crescente da esquerda para a direita. As falhas de enlace que aparecem na figura em linhas pontilhadas são a representação do número máximo de enlaces falhos que cada nó que está difundido pode apresentar, com exceção do nó com nível de segurança 1 o qual pode apresentar mais de um enlace falho, porém difunde para no máximo um nó. Estes enlaces e os nós em suas extremidades inferiores não fazem parte da árvore de difusão apesar de estarem representados na figura. Assim, como já explicado,

o processo de difusão consiste em que cada nó difusor selecione o vizinho com o maior nível de segurança que, em cada etapa e sem que haja repetição, receberá a mensagem encaminhada, até que não haja mais nenhum vizinho a ser atingido pelas dimensões que este nó é responsável. Desta forma, o vizinho com o menor nível de segurança (maior que 0) será aquele que receberá a mensagem na última difusão do nó, se tornando assim uma folha na árvore de difusão. Uma vez que os nós selecionados recebem a mensagem, a difusão em seus respectivos subcubos ocorre como se eles fossem os nós fontes para aqueles subcubos. Vale notar que o nó origem pode ter no máximo um enlace falho; de outra forma ele teria nível de segurança 1. Caso este nó apresente um enlace falho, o nó na outra extremidade deste enlace será considerado falho (0-seguro), apresentando o nível de segurança mais baixo entre os vizinhos do nó origem, e por isso será deixado como a última opção no encaminhamento da mensagem (na realidade ele nunca será uma opção). Para os outros nós origem acontece a mesma coisa, considerando que todos eles têm no máximo um enlace falho. Se tiverem mais de um enlace falho, isto significa que possuem nível de segurança 1 e estão encarregados da difusão em um 1-cubo, o que significa um único nó para o qual encaminhar a mensagem. Conclui-se então que, num processo de difusão, os nós não-falhos que não recebem a mensagem são aqueles vistos como falhos (0-seguros), ou seja, os nós não-perfeitos, não sendo possível um nó perfeito não receber a mensagem. ▽

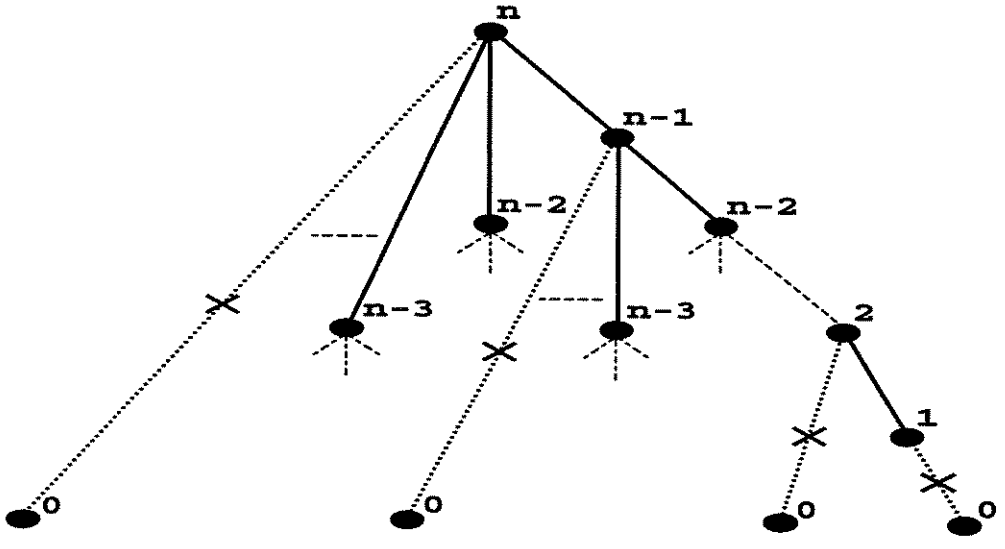

Figura 5.2: Árvore de difusão genérica com falhas de enlace

LEMA 2: Em um hipercubo n-dimensional com menos de n componentes falhos (nós e/ou enlaces), cada nó não-falho tem pelo menos um vizinho seguro.

DEMONSTRAÇÃO:

Suponha, sem perda de generalidade, que um nó não-falho P no hipercubo de 4 dimensões da Fig. 5.3(a) tenha um enlace falho. No máximo $n - 2 = 2$ outras falhas são permitidas em tal hipercubo. Suponha que todas elas sejam enlaces falhos e que elas possam ser convertidas em nós falhos considerando que um dos nós das extremidades de cada um desses enlaces seja falho. Neste caso, o nó que compartilha o enlace falho com P será um dos escolhidos. Esta situação de falha é mais drástica e engloba a situação original, não importando quais nós sejam escolhidos para serem os defeituosos, já que um nó falho pode ser visto como um nó que apresenta todos os seus enlaces falhos. A Fig. 5.3(b) ilustra esta generalização. No artigo [JWU 95] Jie Wu demonstra que em qualquer hipercubo

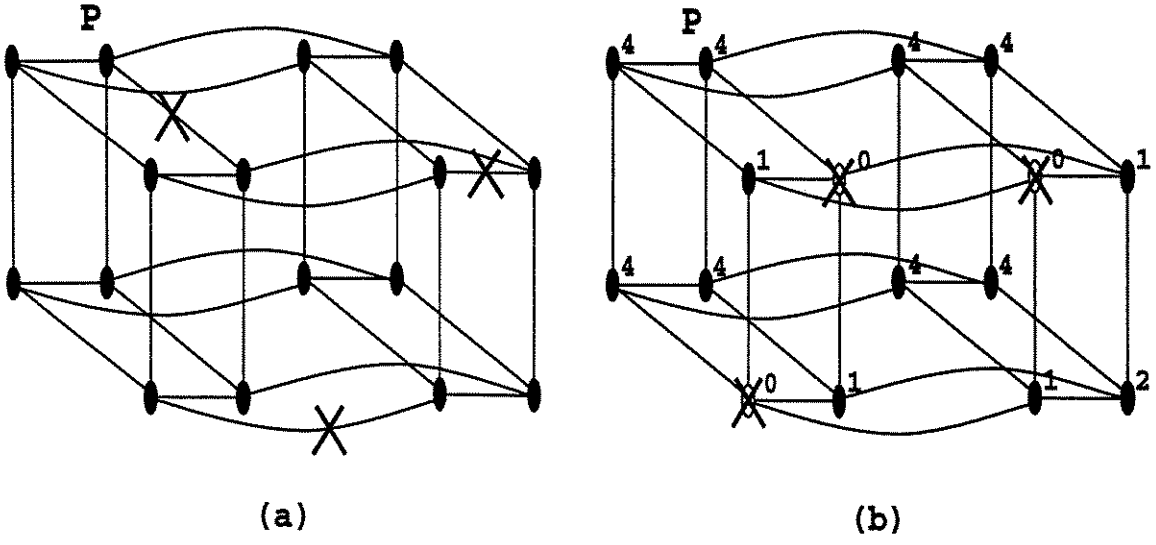

Figura 5.3: (a) Hipercubo com enlaces falhos. (b) Enlaces falhos mapeados em nós falhos.

n-dimensional com menos de n nós falhos (e somente nós falhos), haverá sempre um nó vizinho seguro para um nó não-falho. Assim sendo, no exemplo acima, P sempre terá um vizinho seguro, não importando a distribuição dos enlaces e nós falhos pelo sistema. A Fig. 5.4(a) ilustra a situação mais crítica com P rodeado de enlaces falhos. Nota-se que

mesmo nesta situação, mapeando-se os enlaces falhos em nós falhos (Fig. 5.4(b)), o nó P ainda apresenta um vizinho seguro. De uma forma mais geral, pode-se dizer que, mesmo que cada enlace falho seja convertido em um nó falho em um hipercubo n-dimensional com até $n-1$ falhas, é garantido que todo nó não-falho em tal hipercubo terá pelo menos um vizinho seguro. $\triangledown$

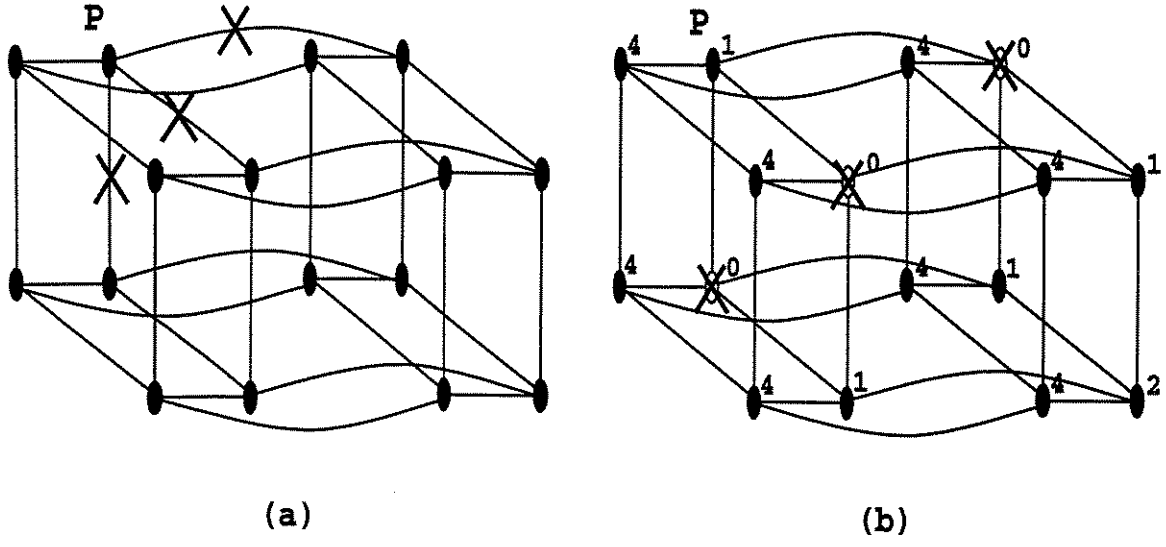

Figura 5.4: (a) Nó P em situação crítica de falha. (b) Enlaces falhos mapeados em nós falhos.

LEMA 3: Em um hipercubo n-dimensional com menos de n componentes falhos (nós e/ou enlaces), cada nó não-falho tem pelo menos um vizinho perfeito.

DEMONSTRAÇÃO:

A demonstração desse lema vem diretamente do fato de que todo nó tem n vizinhos e há no máximo $n-1$ enlaces e/ou nós falhos no hipercubo. Sabendo-se da estrutura do hipercubo que dois vizinhos de um nó nunca compartilham um mesmo enlace, então os vizinhos de um nó A, por exemplo, não podem compartilhar um enlace falho, o que implica que um enlace falho pode tornar não-perfeito um único vizinho de A. Um nó falho não torna nenhum vizinho de A não-perfeito. Daí, somente $n-1$ vizinhos de A podem ser não-perfeitos ou falhos, sobrando pelo menos um vizinho perfeito. $\triangledown$

5.3 Estrutura e lógica de funcionamento do *ESLAB*

Nesta seção é detalhado o algoritmo *ESLAB* baseado no conceito estendido de níveis de segurança. Este algoritmo utiliza este novo conceito em conjunto com uma estratégia simples de monitoramento dos nós não-perfeitos. Todo o processo de difusão é guiado pelos níveis de segurança assim como descrito na Seção 5.2.2. Caso seja detectado que um ou mais nós não-perfeitos não tenham recebido a mensagem durante as n (ou $n+1$, dependendo do nó origem ser ou não seguro) etapas normais do processo, uma etapa extra é realizada para que estes nós a recebam.

O algoritmo *ESLAB* assume que os nós já tenham conhecimento dos níveis de segurança de seus vizinhos e possuam uma lista dos vizinhos perfeitos e não-perfeitos. Eles também conhecem os seus próprios estados. A informação da configuração de falhas local necessária nesta solução é adquirida utilizando-se o algoritmo *DETECT* proposto no Cap. 4. A idéia principal por trás do algoritmo proposto e seus passos de execução são descritos abaixo.

- Se o nó origem é seguro, ele inicia a difusão da mensagem seguindo o método que se baseia no nível de segurança para tratar enlaces falhos; se o nó origem for não-seguro, ele envia a mensagem a um vizinho seguro, cuja existência é garantida pelo LEMA 2, e este então inicia a difusão.

- Para cada nó não-falho, se ele é um nó não-perfeito, há a possibilidade dele não receber a mensagens durante as etapas convencionais do processo. Assim, uma etapa extra pode ser necessária para que a mensagem seja difundida a ele após as etapas iniciais. Como mostra o LEMA 1, apenas nós não-perfeitos têm a possibilidade de não receber a mensagem. Caso se confirme que este nó não-perfeito não recebeu a mensagem, ele a receberá na etapa extra a partir de um vizinho perfeito cuja existência é garantida pelo LEMA 3. Caso ele tenha recebido a mensagem durante as etapas convencionais do processo, ele deve informar este fato aos seus vizinhos perfeitos. Não há necessidade, entretanto, de fazer tal notificação aos nós de quem ele tenha recebido a mensagem ou para quem tenha encaminhado a mesma.

- Para cada nó não-falho, se ele é um nó perfeito, ele deve difundir no passo extra a mensagem aos seus vizinhos não-perfeitos que não a tenham recebido. Para evitar redundância, isto é, um nó receber a mensagem difundida de mais de um vizinho, cada nó perfeito difunde a mensagem apenas para aqueles vizinhos não-perfeitos dos quais ele é o vizinho perfeito conectado pela dimensão mais baixa. Esta não é uma

tarefa difícil de ser realizada considerando-se que todos os nós possuem informação sobre a configuração de falhas local de seus vizinhos.

TEOREMA 1: Em um hipercubo n-dimensional com menos de n falhas (enlaces e/ou nós), o algoritmo *ESLAB* permite que qualquer nó não-falho realize uma operação de difusão com sucesso em no máximo $n + 2$ etapas.

DEMONSTRAÇÃO:

O LEMA 2 mostra que todo nó não-falho possui pelo menos um vizinho seguro. Desta forma, se um nó não-seguro deseja iniciar uma difusão, ele pode encaminhar a mensagem a um vizinho seguro. Isto constitui a primeira etapa do processo. Após o nó seguro ter iniciado a difusão, seguem-se as n etapas convencionais de uma difusão baseada nos níveis de segurança. Após isso, é possível que alguns nós não-falhos tenham deixado de receber a mensagem devido a problemas de enlaces falhos, assim como mostra o LEMA 1. Se isto tiver ocorrido, uma etapa extra é necessária para que aqueles nós recebam a mesma, totalizando $n+2$ etapas. Todos os nós que porventura não tenham recebido a mensagem durante as etapas regulares têm pelo menos um vizinho perfeito (LEMA 3), o qual certamente recebeu a mensagem (LEMA 1). Conhecendo a configuração de falhas de seus vizinhos e tendo recebido uma notificação daqueles não-perfeitos que já tenham recebido a mensagem, os nós perfeitos sabem exatamente para quais nós encaminhar a mesma na etapa final. Eles seguem uma estratégia que não somente os permite atingir os nós desejados, mas também o fazem de uma forma não redundante. $\bigtriangledown$

O pseudo-código para o algoritmo *ESLAB* é exibido no Apêndice B.2. Para melhor entênde-lo, considere-se o exemplo mostrado na Fig. 5.5. O hipercubo de 4 dimensões possui 3 elementos falhos. Duas dessas falhas são de nó (1001 e 0101) e uma de enlace (1*00). Os níveis de segurança são exibidos entre parênteses logo acima dos nós. O nó 1101 é o nó origem da difusão apresentada na figura. Ele é um nó não-seguro e por isso não é capaz de iniciar uma difusão com garantia de sucesso. Assim, ele envia a mensagem pela dimensão 0 ao seu vizinho seguro 1100, que passa a ser então o novo nó origem da difusão. O critério de desempate utilizado aqui em caso de vizinhos com níveis de segurança iguais, é escolher aquele conectado pela dimensão mais baixa. Por isso o nó 1101 escolheu

o nó 1100 (conectado pela dimensão 0) ao invés de ter escolhido o nó 1111 (conectado pela dimensão 1), ambos com nível de segurança 4.

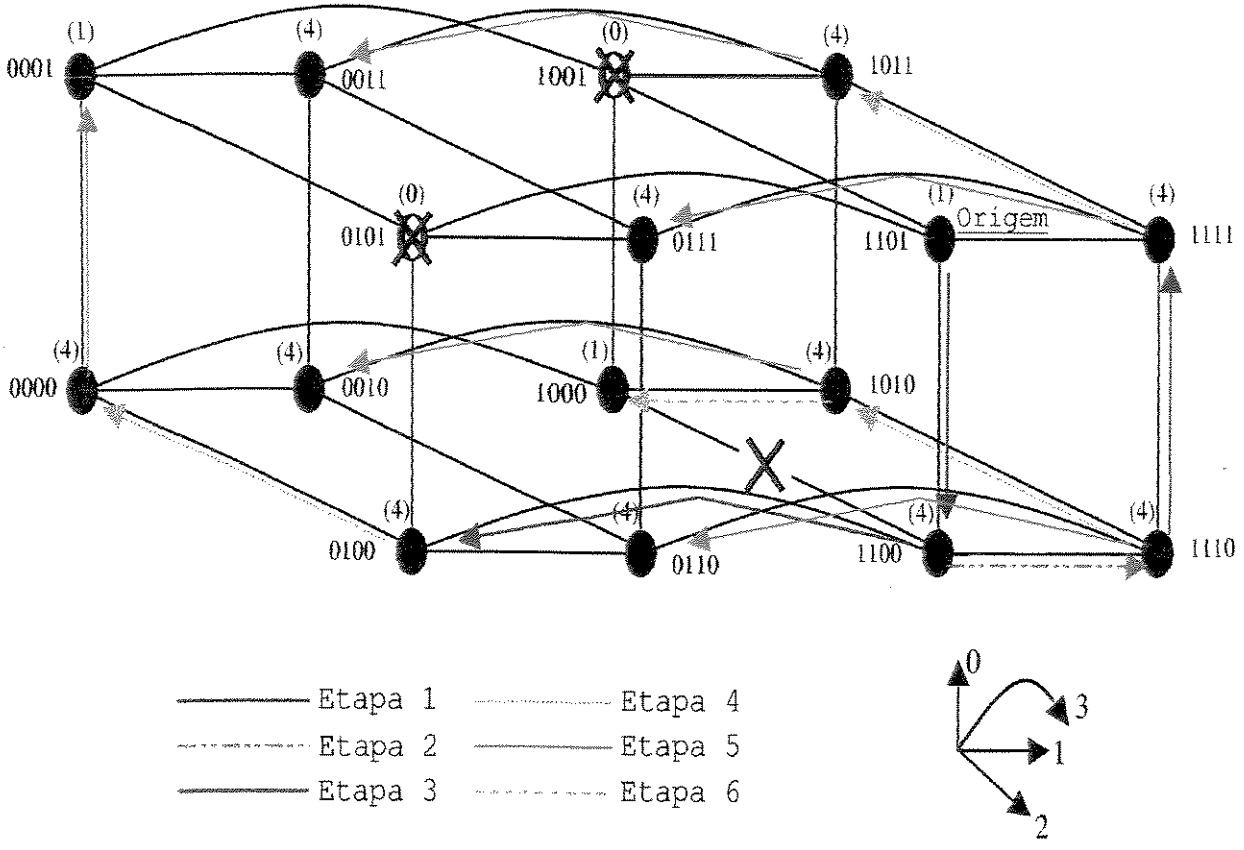

Figura 5.5: Difusão em um hipercubo de 4 dimensões com 3 falhas utilizando o algoritmo *ESLAB*

O nó 1100 inicialmente (etapa 2) envia a mensagem ao nó 1110 pela dimensão 1. Na terceira etapa, o nó 1100 difunde pela dimensão 3 atingindo o nó 0100, enquanto o nó 1110 difunde pela dimensão 0 atingindo o nó 1111. Na quarta etapa, o nó 1100 não reenvia a mensagem ao nó 1101, pois este já possui a mesma. Nesta etapa, os nós 1110, 0100 e 1111 enviam a mensagem aos nós 1010, 0000 e 1011, respectivamente, pela dimensão 2. Na penúltima etapa da difusão, os nós 1110, 1111, 1011 e 1010 enviam a mensagem aos nós 0110, 0111, 0011 e 0010, respectivamente, pela dimensão 3, enquanto o nó 0000 difunde a mesma ao nó 0001 pela dimensão 0. Neste momento, os nós não-perfeitos que receberam a mensagem deveriam enviar uma mensagem de notificação, a respeito deste fato, aos seus vizinhos perfeitos que não tenham conhecimento disto ainda. Os únicos nós não-perfeitos do sistema são os nós 1000 e 1100. O nó 1100 não necessita fazer qualquer notificação já

que todos os seus vizinhos perfeitos se comunicaram com ele durante o processo, através de envio ou recebimento da mensagem, e sabem que ele já possui a mesma. Nota-se que apesar do nó 1000 não ser falho ele não recebeu a mensagem devido a um enlace falho que impediu o nó 1100 de enviar a mesma a ele. Por isso, ele não faz qualquer notificação aos seus vizinhos perfeitos. Os vizinhos perfeitos do nó 1000 são os nós 1010 e 0000. Estes nós apresentam os seguintes vetores de falhas e matriz F construídos durante a execução do algoritmo *DETECT* (ver Seção 4.3.2).

$fault = [\ 0020\ ]$

$$F = \begin{bmatrix} 0010 \\ 0101 \\ 0020 \\ 0000 \end{bmatrix} \begin{matrix} \text{dimensão 0} \\ \text{dimensão 1} \\ \text{dimensão 2} \\ \text{dimensão 3} \end{matrix}$$

nó 1010

$fault = [\ 2000\ ]$

$$F = \begin{bmatrix} 1100 \\ 0000 \\ 2001 \\ 0101 \end{bmatrix} \begin{matrix} \text{dimensão 0} \\ \text{dimensão 1} \\ \text{dimensão 2} \\ \text{dimensão 3} \end{matrix}$$

nó 0000

A partir de seu vetor de falhas, o nó 0000 percebe que possui um vizinho não-perfeito pela dimensão 3 do qual ele não recebeu uma notificação sobre o recebimento da mensagem difundida. Assim, ele verifica sua matriz F na linha correspondente à dimensão 3 para checar se ele é o vizinho perfeito de 1000 pela dimensão mais baixa. Ao fazer esta verificação ele percebe, ao encontrar um 0 na dimensão 1 daquela linha, que existe um outro nó perfeito que é vizinho de 1000 por uma dimensão mais baixa que a dele, e daí ele nada faz. O nó 1010 realiza o mesmo procedimento descobrindo que ele é o nó perfeito vizinho de 1000 pela dimensão mais baixa, e assim o responsável por enviar a mensagem ao mesmo. Desta forma, na sexta e última etapa do processo, o nó 1010, e somente ele, envia a mensagem ao nó 1000 pela dimensão 1.

Considere-se agora o exemplo mostrado na Fig. 5.6. onde o hipercubo de 4 dimensões possui 5 falhas. Utiliza-se neste exemplo mais de $n-1$ falhas para mostrar que o algoritmo *ESLAB* pode funcionar mesmo em uma situação mais adversa, embora isto não seja garantido. Duas dessas falhas são de nó (0111 e 1001) e as outras três são de enlace (0∗01, 10∗0 e 111∗). O nó 1100 é o nó origem da difusão apresentada na figura. Ele é um nó seguro e desta forma capaz de iniciar uma difusão que pode atingir todos os demais nós não-falhos. O nó 1100 inicialmente envia a mensagem ao nó 1110 pela dimensão 1. Na segunda etapa, o nó 1100 difunde pela dimensão 3 atingindo o nó 0100, enquanto o nó 1110 difunde pela dimensão 2 atingindo o nó 1010. Os nós 1100 e 1010 então, na terceira etapa,

difundem a mensagem pela dimensão 0 alcançando os nós 1101 e 1011, respectivamente. Nesta mesma etapa, o nó 1110 envia a mensagem ao nó 0110 pela dimensão 3 enquanto o nó 0100 difunde pela dimensão 2 atingindo o nó 0000. Na penúltima etapa da difusão, o nó 1100 difunde pela dimensão 2 atingindo o nó 1000, os nós 1011 e 1010 enviam a mensagem aos nós 0011 e 0010, respectivamente, pela dimensão 3, e os nós 0100 e 0000 difundem a mesma aos nós 0001 e 0101 pela dimensão 0. Nota-se que o nó 1111, apesar de não ser falho, deixou de receber a mensagem difundida. Na quinta e última etapa do processo então, ele receberá a mesma a partir do nó 1101 que é o seu vizinho perfeito pela dimensão mais baixa.

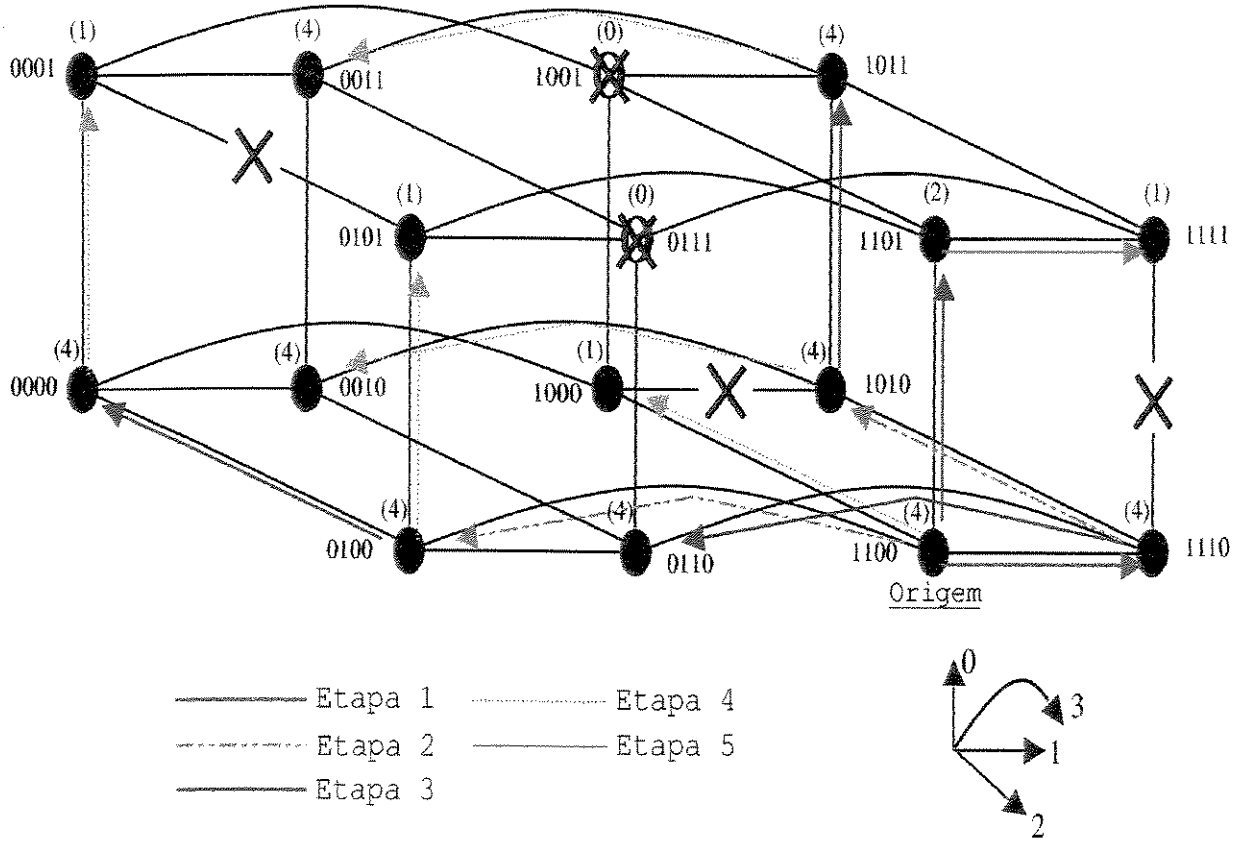

Figura 5.6: Difusão em um hipercubo de 4 dimensões com 5 falhas utilizando o algoritmo *ESLAB*

5.4 Testes e análise de desempenho

Nesta seção, faz-se uma análise do desempenho do algoritmo *ESLAB* e compara-se o mesmo com outros algoritmos propostos para resolver um problema similar.

A complexidade de tempo do processo de preparação do sistema para que o *ESLAB* possa ser executado com sucesso está relacionada com o processo de detecção e localização dos componentes defeituosos presentes no sistema, armazenamento das informações de configuração de falhas nos nós e definição dos níveis de segurança dos mesmos. Este esforço inicial é necessário apenas uma vez, não importando quantas operações de difusão são realizadas durante a execução de uma aplicação distribuída. As tarefas que esta etapa compreende podem ser realizadas quando a máquina é reinicializada ou quando uma nova falha é detectada ou algum componente que estava defeituoso é reparado. Embora este processo de inicialização do sistema não seja parte do algoritmo *ESLAB*, o levantamento de sua complexidade é necessário para efeito de comparação com os outros algoritmos. A complexidade total estimada para realizar a tarefa de localização das falhas é aquela apresentada na Seção 4.4.4, já que se está utilizando o algoritmo *DETECT* para este propósito, mais o tempo gasto na etapa que compreende o envio de notificação por parte dos nós não-perfeitos aos seus vizinhos que atualizam seus vetores de falha com esta informação. Com relação à definição dos níveis de segurança, a complexidade é aquela descrita na Seção 3.2.4. Estas complexidades são:

- Localização das falhas: $2n^2 + 11n + 5$ *passos*
- Definição dos níveis de segurança: $n^3 + n^2 + 4n - 6$ *passos*

Podemos então afirmar que a complexidade para o esforço inicial é de $O(n^3)$. Os valores apresentados acima foram calculados utilizando-se a mesma métrica utilizada na Seção 3.2 e que será utilizada na análise do algortimo *ESLAB*.

Agora, apresenta-se a complexidade do algoritmo *ESLAB*. Dado que para ele são necessárias no máximo $n+2$ etapas para finalizar uma difusão, e que o esforço computacional para cada uma delas não é necessariamente o mesmo, analisa-se cada uma separadamente. A primeira etapa (que pode ou não existir) é aquela em que um nó não-perfeito, desejando iniciar uma difusão, envia a mensagem a um vizinho perfeito. As n etapas seguintes podem ser consideradas iguais em termos de complexidade, pois elas seguem um padrão similar. A última etapa (se existir) compreende nós não-perfeitos enviando mensagens de

notificação a seus vizinhos perfeitos e então vizinhos perfeitos enviando a mensagem àqueles não-perfeitos que não a receberam nas etapas anteriores. Assim, a complexidade para cada uma das partes analisadas de acordo com o código utilizado na implementação do algoritmo é dada abaixo.

- Primeira etapa: $\frac{n}{2} + 6$ *passos*
- As n etapas seguintes: $\frac{9n}{2} + 21$ *passos*
- Última etapa: $n^2 + 3n$ *passos*

A complexidade então do algoritmo *ESLAB* é $O(n^2)$. A complexidade do processo como um todo incluindo o esforço inicial é da ordem de grandeza da complexidade do esforço inicial, isto é, $O(n^3)$.

Nas próximas seções são feitas análises mais profundas sobre o desempenho de *ESLAB* baseadas nos testes realizados, fazendo-se um estudo sobre os dados coletados e comparando-se estes resultados obtidos com os de outros algoritmos.

5.4.1 O Ambiente, as limitações e os algoritmos das simulações

O ambiente utilizado nos testes de *ESLAB* foi exatamente o mesmo usado para os testes do algoritmo *DETECT*, assim como apresentado na Seção 4.3.3. Por ser um hipercubo de 6 dimensões, o n-Cube2 utilizado na implementação foi aproveitado para simular hipercubos de dimensões menores. Assim, os testes foram feitos para subcubos de 3, 4, 5 e 6 dimensões, todos eles no mesmo n-Cube2 de 6 dimensões. Com isso, foi possível analisar o desempenho dos algoritmos quando o tamanho do hipercubo varia. As falhas, assim como nos testes do algoritmo *DETECT*, também foram geradas por software passando-se um vetor contendo todos os identificadores dos nós e enlaces supostamente falhos como parâmetro para o programa. Esta foi uma das limitações dos testes já que não foi possível, por hardware, interromper o funcionamento de nós ou enlaces específicos. O fato dos nós terem de consultar este vetor, gerou atrasos na execução do programa que não existiriam em um caso real. Estes atrasos porém, além de baixos, só foram sentidos na execução do processo inicial de detecção e localização das falhas, já que é só naquele momento que se precisa saber quem realmente é falho ou não para se gerar os vetores locais de falha e calcular os níveis de segurança. Não fosse por isso, a utilização de passagem de parâmetros para

simular as falhas se mostraria uma solução inadequada para o problema. Outra limitação está relacionada com a leitura dos tempos de execução dos algoritmos. O Apêndice A.2 aborda este problema com detalhes.

Os algoritmos escolhidos para comparação com *ESLAB* foram o de Jie Wu (1995) [JWU 95] e o de Lee & Hayes (1992) [LEE 92]. Eles foram escolhidos não só pelo desempenho satisfatório que apresentaram quando comparados com outros algoritmos, como visto na Seção 3.2.8, mas também por apresentarem uma estratégia de difusão similar ao do *ESLAB*. De certa forma, a estratégia utilizada no algoritmo *ESLAB* pode ser vista como complementar às estratégias propostas por Wu e por Lee e Hayes. Jie Wu, com a estratégia dos níveis de segurança, complementou a idéia inicial dos nós não-seguros criada por Lee e Hayes, e já o conceito estendido dos níveis de segurança, apresentado neste trabalho, é uma complementação à estratégia dos níveis de segurança apresentada por Wu.

Para que se tivesse uma comparação mais justa entre *ESLAB* e os outros dois algoritmos, implementou-se[2] os três utilizando a mesma linguagem (C) e a mesma técnica de programação para que nenhum deles fosse favorecido ou prejudicado. Os algoritmos de Jie Wu (1995) e Lee & Hayes (1992) foram implementados de acordo com os pseudo-códigos apresentados nos artigos onde os mesmos foram propostos. Além disso, pelo fato de Jie Wu (1995) e Lee & Hayes (1992) tratarem tipos e quantidades de falhas diferentes, comparou-se cada um deles com *ESLAB* de forma independente. Os mesmos casos de falhas utilizados no teste de Jie Wu (1995) foram utilizados em *ESLAB* e, da mesma forma, os casos de falhas utilizados no teste de Lee & Hayes foram também utilizados em *ESLAB*. Assim, na hora da comparação com cada um dos dois algoritmos, somente foram utilizados os dados colhidos de *ESLAB* referentes aos casos de falha do algoritmo comparado. O resultado destas comparações é apresentado nas seções seguintes em forma de tabelas, gráficos e comentários. Vale dizer que parte dos teste realizados com *ESLAB* foi monitorada para comprovar que ele realmente, sob as condições estabelecidas, fazia com que todos os nós não-falhos do sistema recebessem a mensagem difundida.

5.4.2 *ESLAB versus* LEE & Hayes 92

Como visto no Cap. 3, o algoritmo de Lee & Hayes (1992) trata tanto enlaces como nós falhos. O número máximo de nós falhos tolerados é $\lceil \frac{n}{2} \rceil$. Quando há a presença de enlaces falhos o número máximo de elementos falhos suportados cai para $\lceil \frac{n}{2} \rceil - 1$. Assim, de

[2] Os pseudo-códigos dos algoritmos implementados encontram-se no Apêndice B.

acordo com as tabelas do Apêndice A.3, que apresentam o número máximo de combinações de falhas possíveis para os subcubos de 3 a 6 dimensões, o número de testes realizados com o algoritmo de Lee & Hayes (1992) foram 41 para o subcubo de 3 dimensões, 153 para o subcubo de 4 dimensões, 10557 para o subcubo de 5 dimensões e 71974 para o subcubo de 6 dimensões. Todas essas combinações de falhas, para os 4 diferentes subcubos, foram também testadas utilizando-se *ESLAB*. Para se ter uma noção do tempo gasto para realizar os testes, apenas para o subcubo de 6 dimensões, por exemplo, que demandou 71974 execuções de cada um dos algoritmos, gastou-se mais de 5 dias. As Tabelas 5.1 e 5.2 apresentam o valor médio, o desvio padrão e o erro padrão para cada hipercubo para os processos inicial e de difusão, respectivamente, do algortimo de Lee & Hayes (1992). As Tabelas 5.3 e 5.4 apresentam o mesmo tipo de informação para os dados colhidos dos testes com *ESLAB*.

Os valores de desvio e erro padrão são importantes no sentido de se avaliar quão perto ou longe da média estão os resultados obtidos em cada teste. Caso o desvio e o erro padrão sejam altos, isto pode indicar algum tipo de comportamento irregular por parte dos algoritmos, ou seja, podem indicar a existência de alguns casos de falha para os quais o algoritmo se torna mais rápido ou mais lento. Observando-se os resultados apresentados nas tabelas, percebe-se que os dois algoritmos se comportam de maneira regular com a variação das falhas.

Tabela 5.1: Esforço Inicial para Lee & Hayes 92

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	8519	480	75
4	17392	537	84
5	24148	1602	31
6	40487	1968	7

A Tabela 5.5 apresenta o esforço total médio para o processo de difusão completo para Lee & Hayes 92 e *ESLAB* e a Fig. 5.7 apresenta estes dados de forma gráfica.

Percebe-se claramente através destes dados que o algoritmo *ESLAB* não só trata mais falhas (quase o dobro) que Lee & Hayes (1992), como também é mais rápido que ele. No gráfico apresentado na Fig. 5.7 *ESLAB* apresenta um comportamento linear com a variação do tamanho do hipercubo, enquanto que a curva para Lee & Hayes 92 se aproxima

Tabela 5.2: Esforço de difusão para Lee & Hayes 92

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	1027	223	35
4	1186	167	26
5	1829	461	4
6	2263	439	2

Tabela 5.3: Esforço Inicial para *ESLAB*

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	7049	518	81
4	11466	535	84
5	16159	1314	13
6	21081	1611	6

Tabela 5.4: Esforço de difusão para *ESLAB*

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	1198	391	61
4	1863	478	75
5	2286	545	5
6	2613	580	2

Tabela 5.5: Esforço completo no processo de difusão

N^o de dimensões	*ESLAB* (μs)	Lee & Hayes 92 (μs)
3	8247	9546
4	13329	18577
5	18445	25978
6	23694	42750

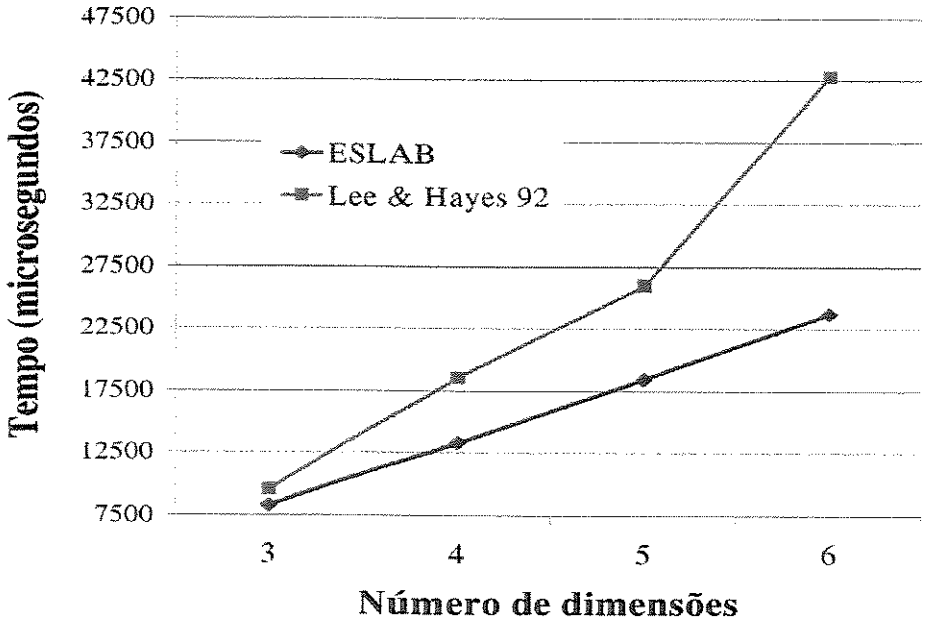

Figura 5.7: Esforço total - *ESLAB* versus Lee & Hayes 92

a uma parábola. Nota-se que para um hipercubo de 3 dimensões os dois algoritmos têm desempenhos praticamente iguais, enquanto que para um hipercubo de 6 dimensões *ESLAB* é quase duas vezes mais rápido.

Até o momento, considerou-se apenas o caso de uma única difusão no processo. Como já visto, uma aplicação distribuída pode demandar uma série de difusões durante a sua execução. Caso esta aplicação utilize um desses dois algoritmos, o esforço inicial será necessário apenas uma vez para a realização de todas as difusões. Assim sendo, é importante verificar o comportamento apenas do algoritmo de difusão em si. A Fig. 5.8 apresenta as curvas geradas a partir dos dados das Tabelas 5.2 e 5.4. É possível notar que apesar de Lee & Hayes (1992) ser mais rápido que *ESLAB* no que se refere apenas ao esforço de difusão propriamente dito, a diferença entre o desempenho dos dois diminui com o aumento do hipercubo. Considerando o hipercubo de 4 dimensões, que é aquele para o qual a diferença de desempenho é a mais acentuada, somente após nove difusões o algoritmo de Lee & Hayes (1992) passa a ser mais vantajoso que *ESLAB* em termos de desempenho. Este fato é representado na Fig. 5.9, cujos dados foram gerados computando-se para cada nova difusão, e para cada algoritmo, apenas o esforço relativo à difusão, sendo que o esforço inicial só é computado uma única vez.

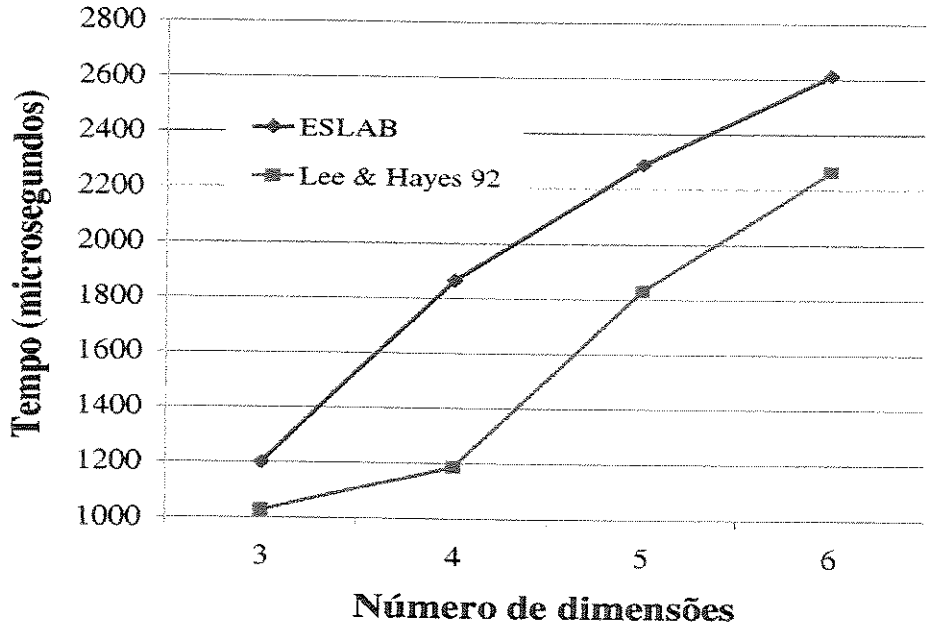

Figura 5.8: Esforço só de difusão - *ESLAB* versus Lee & Hayes 92

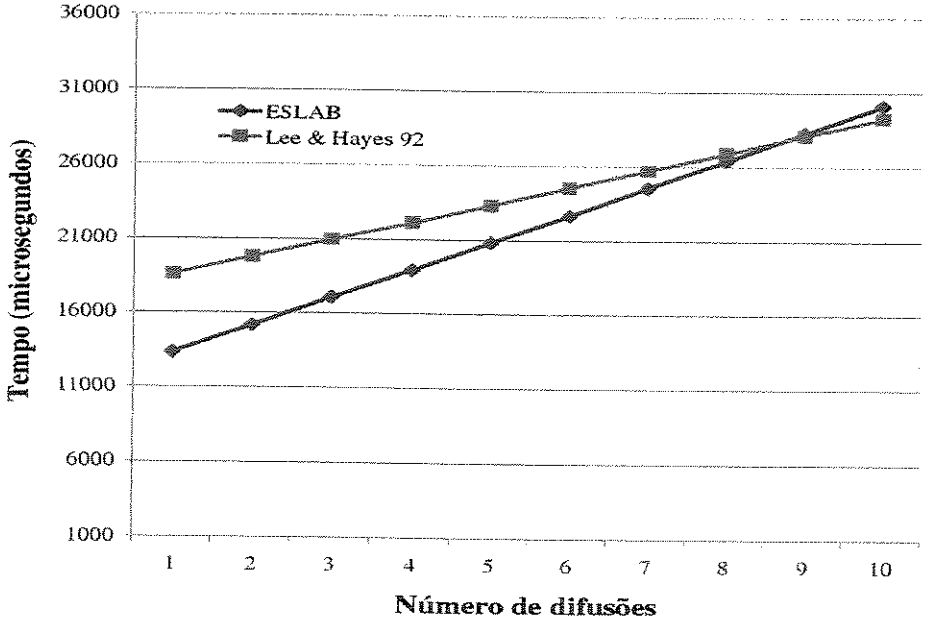

Figura 5.9: Desempenho para um hipercubo de 4 dimensões quando o número de difusões varia

5.4.3 *ESLAB versus* Jie Wu 95

Como visto anteriormente, o algoritmo de Jie Wu (1995) trata apenas nós falhos. O número máximo de nós falhos tolerados é $n-1$. De acordo com as tabelas do Apêndice A.3, o número máximo de combinações de falhas de nós até um máximo de $n-1$ falhas para os subcubos de 3, 4, 5 e 6 dimensões, é, respectivamente, de 29, 576, 36457 e 7666240. Foram realizados testes com todos os casos de falhas para os subcubos até 5 dimensões. Devido à grande quantidade de combinações de casos de falhas para o hipercubo de 6 dimensões e baseando-se no tempo gasto com os outros testes (alguns dias), decidiu-se testar apenas um subconjunto destes casos de falha. Este subconjunto foi composto por todos os casos de falhas até 2 falhas, 10% dos casos de falhas para 3 falhas e 1% dos casos de falhas para 4 e 5 falhas no sistema, num total de 82233 casos de falhas. Acredita-se que este subconjunto seja representativo em termos das variações de quantidade e distribuição das falhas no hipercubo já que vários testes foram realizados com diferentes tipos de falhas e distribuição pelo hipercubo e o erro padrão para estes casos foi bem baixo. Além disso, todos os subcasos de combinações de nós e enlaces falhos estão representados nos conjuntos de casos de teste escolhidos para 3, 4 e 5 falhas, ainda que não se tenha garantido que eles tenham uma representação equilibrada, isto é, um percentual de teste equivalente para cada subcaso. Todas essas combinações de falhas, para os 4 diferentes hipercubos, foram também testadas utilizando-se *ESLAB*. As Tabelas 5.6 e 5.7 apresentam o valor médio, o desvio padrão e o erro padrão para cada hipercubo para os processos inicial e de difusão, respectivamente, do algortimo de Jie Wu (1995). As Tabelas 5.8 e 5.9 apresentam o mesmo tipo de informação para os dados colhidos dos testes com *ESLAB*.

Assim como nos testes com os casos de falha de Lee & Hayes (1992), os resultados apresentados nestas tabelas indicam que os dois algoritmos se comportam de maneira regular com a variação das falhas. A Tabela 5.10 apresenta o esforço total médio para o processo de difusão completo para Jie Wu (1995) e *ESLAB* e a Fig. 5.10 apresenta o gráfico gerado a partir destes dados.

É interessante notar que além de tratar um tipo de falha que é mais provável de ocorrer e mais difícil de ser tratado, o algoritmo *ESLAB* é apenas um pouco mais lento que Jie Wu (1995) que tolera a mesma quantidade de falhas que *ESLAB*, mas só trata nós falhos. Essa diferença de tempo, por exemplo, é de pouco mais de 50% para o hipercubo de 6 dimensões. Considerando-se haver várias difusões em uma aplicação que venha utilizar um destes dois algoritmos, essa diferença de tempo chega a cair para 25% para o caso de 50 difusões. A curva que representa a diferença de desempenho dos dois algoritmos em função

Tabela 5.6: Esforço Inicial para Jie Wu 95

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	3329	82	15
4	5832	4	1
5	8946	90	1
6	12873	197	1

Tabela 5.7: Esforço de difusão para Jie Wu 95

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	844	172	32
4	1218	58	11
5	1634	220	1
6	2049	243	1

Tabela 5.8: Esforço Inicial para *ESLAB*

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	6738	211	39
4	10507	6	1
5	14959	94	1
6	20228	205	1

Tabela 5.9: Esforço de difusão para *ESLAB*

N^o de dimensões	Médio (μs)	Desvio Padrão (μs)	Erro Padrão (μs)
3	1028	218	41
4	1444	85	16
5	2038	354	2
6	2479	332	1

Tabela 5.10: Esforço completo no processo de difusão

N^o de dimensões	*ESLAB* (μs)	Jie Wu 95 (μs)
3	7766	4174
4	11951	7050
5	16997	10579
6	22706	14922

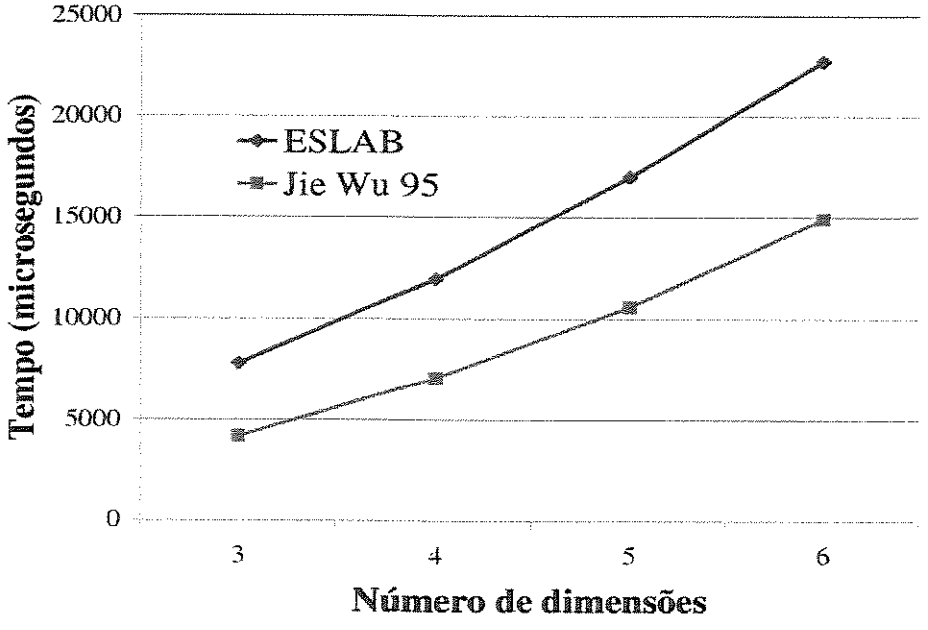

Figura 5.10: Esforço total - *ESLAB* versus Jie Wu 95

do número de difusões é exibida na Fig. 5.11. Os dados apresentados na Fig. 5.11 foram gerados dividindo-se a diferença dos tempos obtidos para *ESLAB* e Jie Wu (1995) pelos tempos obtidos para Jie Wu (1995). Estes tempos foram obtidos computando-se para cada nova difusão, e para cada algoritmo, apenas o esforço relativo à difusão, sendo que o esforço inicial só é computado uma vez.

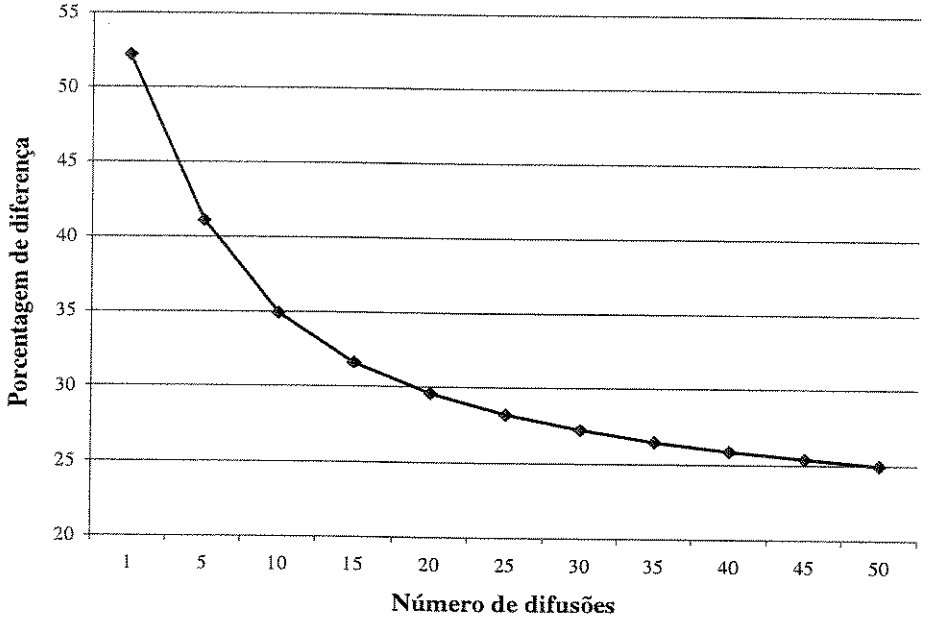

Figura 5.11: Diferença no desempenho para um hipercubo de 6 dimensões quando o número de difusões varia

5.4.4 Desempenho geral de *ESLAB*

A partir dos resultados das comparações feitas de *ESLAB* com outros algoritmos propostos para resolver um problema similar, conclui-se que seu desempenho é bem satisfatório visto a quantidade e os tipos de falhas que ele tolera. Observando-se, por exemplo, os resultados do teste de *ESLAB* no hipercubo de 6 dimensões para os casos de falhas referentes ao algoritmo de Jie Wu (1995), não se nota qualquer tipo de padrão de variação no desempenho de acordo com o aumento do número de falhas. Isto indica que o *ESLAB* é bem mais sensível à distribuição das falhas no sistema do que à quantidade das mesmas. Para avaliar o seu desempenho com relação ao tipo de falha presente no sistema, testou-se 5000

casos de falhas somente de nós, gerados aleatoriamente, e 5000 casos de falhas somente de enlaces, também gerados aleatoriamente. Nestas duas baterias de testes o número de falhas era fixo em 5 e o hipercubo utilizado foi o de 6 dimensões. O resultado pode ser visto na Fig. 5.12.

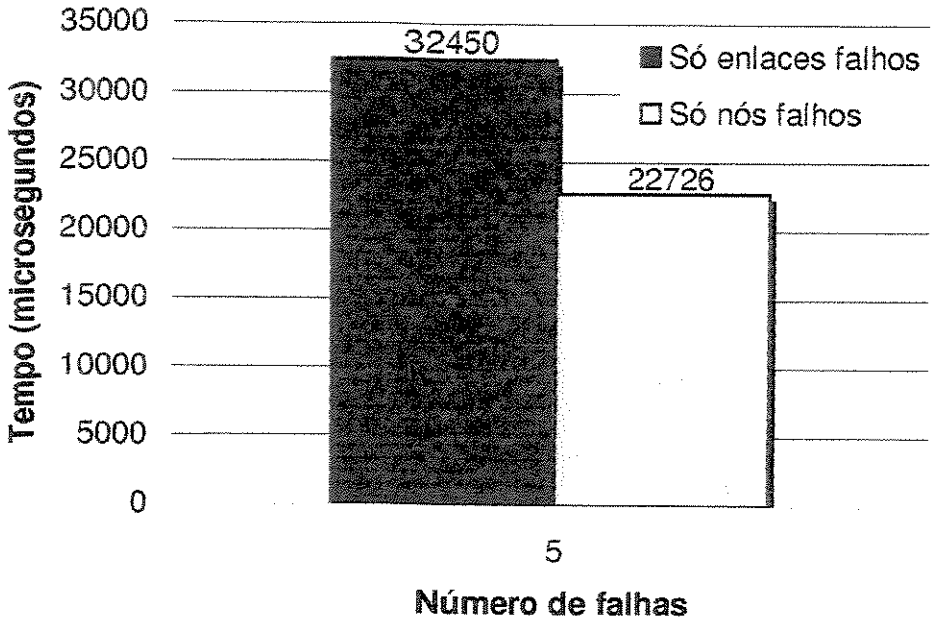

Figura 5.12: Desempenho de *ESLAB* de acordo com o tipo de falha para hipercubo de 6 dimensões

Nota-se que *ESLAB* é razoavelmente sensível ao tipo de falha presente no sistema. A média dos valores de tempo registrados para o caso de apenas enlaces falhos presentes no sistema é 43% maior que a média registrada para o caso de apenas nós falhos no hipercubo. Essa diferença se explica pelo fato de que somente quando existem enlaces falhos no sistema é que todo o processo de *ESLAB* de monitoramento dos nós não-perfeitos acontece. Quando não existem enlaces falhos, também não existem nós não-perfeitos e não há a necessidade de uma etapa extra de difusão no processo. Neste caso, haverá no máximo $n + 1$ etapas no processo de difusão. Esta é uma característica interessante de *ESLAB* já que ele só demanda um maior esforço computacional quando realmente é necessário.

A tabela de comparações entre algoritmos de difusão de mensagens tolerantes a falhas para hipercubos apresentada no Cap. 3, foi refeita para incluir *ESLAB*. A Tabela 5.11 exibe esta inclusão.

Tabela 5.11: Comparação do *ESLAB* com outros algoritmos

Algoritmo / Características	Lee & Hayes 1992	J. Bruck 1994	Jie Wu 1994	Jie Wu 1995	Jie Wu 1996	ESLAB
Trata falhas de Enlace/Nó	**S/S**	S/N	S/N	N/S	S/N	**S/S**
Número máximo de falhas para garantir resultados desejados	$\lceil n/2 \rceil$	$\lceil n/2 \rceil - 1$	**n - 1**	**n-1**	**n - 2**	**n-1**
Número máximo de falhas para ainda ter a chance de obter resultados desejados	n.d.	n - 2	**$n2^{n-1} - 2^n - 1$**	n.d.	n.d.	n.d.
Difusão a partir de qualquer nó	**S**	**S**	**S**	**S**	**S**	**S**
Tipo de informação de configuração de falhas	**informação local**	informação global	informação global	**informação local**	informação global limitada	informação global limitada
Redundância	**N**	**N**	**N**	**N**	**N**	**N**
Estratégia	nó não-seguro	árvore binária	árvore binária / particionamento	níveis de segurança	árvore binária / um nó fonte por nível	níveis de segurança estendidos
Complexidade inicial (em *passos*)	$2n^3 + 5n^2 + 2n$	$2^n n.(3n - 3/4) + (5/2)(n^2 - n - 1)$	$4n^3 + 2n^2$	$n^3 + n^2 + 4n - 6$	**$n^2 + 5n$**	$n^3 + 3n^2 + 15n - 1$
Complexidade apenas da difusão por etapa (em *passos*)	$n^2/2 + 8n + 5$	$2^{n-1} + 9$	**n + 12**	**4n + 6**	$n^3/8 + (21/8)n^2 + 2n + 5$	**9n/2 + 21**
Número máximo de etapas para concluir a difusão	n + 1	**n**	n + 1	n + 1	**n**	n + 2

5.5 Aplicações para *ESLAB*

Várias são as aplicações que podem utilizar um algoritmo de difusão de mensagens tolerante a falhas. Um exemplo seria uma solução distribuída para o problema do caixeiro viajante onde cada nó de um sistema multicomputador executa um mesmo algoritmo porém com parâmetros de entrada diferentes, sendo que, se em um determinado momento um nó encontra um caminho melhor que os até então encontrados, ele difunde este caminho para todos os outros nós que passam a trabalhar este novo caminho utilizando os parâmetros que eles possuem. Caso existam falhas no sistema, é de se esperar que os resultados sejam comprometidos ou que haja uma piora no desempenho da aplicação. Se tal aplicação estiver sendo executada em um hipercubo onde existam falhas de nó e enlace, a Tabela 5.11 indica que *ESLAB* é a melhor opção de escolha para realizar as difusões nesta aplicação, já que trata o maior número de ocorrências e de tipos de falhas dentre os algoritmos criados para a mesma finalidade. Além disso ele o faz com eficiência, apresentando um desempenho bastante satisfatório.

Apesar de *ESLAB* ter sido desenvolvido para ser utilizado em um hipercubo físico tipo o n-Cube2, ou seja, uma máquina especificamente configurada numa estrutura hipercúbica onde os enlaces são canais físicos e os nós estão fortemente acoplados, ele também pode ser utilizado em sistemas logicamente configurados como um hipercubo, os chamados hipercubos lógicos. Estes hipercubos lógicos podem ser construídos a partir de uma rede local, onde os nós (computadores) podem estar alojados numa mesma sala, a partir de uma rede ampla, onde os nós podem estar espalhados pelo campus de uma universidade por exemplo, ou mesmo a partir da Internet. No caso de um hipercubo lógico formado a partir de computadores espalhados pela Internet, o nível de acoplamento dos nós é mínimo, enquanto que em um hipercubo físico ele é máximo.

Tanto para os hipercubos físicos quanto para os lógicos, o conceito de falha em nó é o mesmo, significando um processador ou computador com defeito e que não é capaz de interagir com os outros. Entretanto, o que seriam as falhas de enlace que *ESLAB* deveria tratar no caso de estar sendo utilizado em um hipercubo lógico? Os nós nos hipercubos físicos, em geral, possuem diversos canais de comunicação independentes, um para cada dimensão do hipercubo. Os nós dos hipercubos lógicos, por sua vez, raramente apresentarão mais de um canal de comunicação. Assim, uma falha no canal de comunicação de um nó de um hipercubo lógico implicará na perda completa daquele nó, configurando assim uma falha de nó. Por isso, o conceito de enlace para os hipercubos lógicos deve também ser lógico. O enlace pode ser, por exemplo, o conhecimento por parte de cada nó do sistema

do endereço (físico ou lógico) de seu vizinho. Caso um nó, por qualquer que seja a razão, perca este endereço, então estará configurada uma falha de enlace, já que os dois nós vizinhos não mais conseguirão se comunicar. Um outro conceito para enlace lógico poderia ser a rota utilizada pelas mensagens que trafegam entre dois nós. Supondo que seja possível estabelecer rotas fixas entre dois nós quaisquer, caso haja algum defeito em um *gateway*, por exemplo, utilizado por uma dessas rotas, estaria então configurada uma falha em um enlace. Seja qual o for o conceito utilizado, desde que o número de falhas do sistema seja inferior à dimensão do hipercubo lógico, *ESLAB* pode perfeitamente ser utilizado para realizar difusões tolerantes a falhas.

5.6 Conclusão

Neste capítulo, foi apresentado um algoritmo distribuído e tolerante a falhas para difusão de mensagens em hipercubos defeituosos. Este algoritmo, batizado de *ESLAB*, possui as características mais favoráveis de um algoritmo de difusão tolerante a falhas no que se refere à quantidade de informação sobre a configuração de falhas do sistema que os nós necessitam, a quantidade de falhas e aos tipos de falhas toleradas. Para até $n-1$ falhas em um hipercubo n-dimensional, sejam elas de nós, enlaces ou uma combinação de ambas, *ESLAB* necessita de no máximo $n+2$ etapas para finalizar o processo de difusão, a um custo de tempo de $O(n^2)$.

ESLAB foi implementado e intensamente testado em um hipercubo comercial. Os resultados desses testes foram comparados com os de dois outros algoritmos que também foram implementados e testados na mesma máquina e sob as mesmas condições. Esta comparação revelou um desempenho bastante satisfatório por parte de *ESLAB*. Um ponto que necessita de um trabalho futuro em *ESLAB* está na determinação do tempo que os nós perfeitos devem esperar, na última etapa do processo de difusão, antes de decidirem se devem ou não enviar a mensagem ao seus vizinhos não-perfeitos. Não é simples ajustar este tempo, mas um mau ajuste no mesmo provocará no máximo redundâncias (nós perfeitos enviando mensagens a nós não-perfeitos que já a receberam) ou um leve acréscimo no tempo de execução do algoritmo, não comprometendo de forma alguma o seu funcionamento correto.

Capítulo 6

Conclusões Finais e Trabalhos Futuros

6.1 Conclusões

Este estudo abordou as questões da detecção e localização de falhas e da difusão de mensagens em hipercubos defeituosos.

No que diz respeito à detecção e localização de falhas, foram apresentados os aspectos mais relevantes sobre o assunto e os métodos mais tradicionais de se realizar tal tarefa em sistemas multicomputadores. Estes métodos e os algoritmos propostos mais recentemente para implementá-los foram analisados, descobrindo-se sua inadequação para fornecer informações sobre falhas do sistema a algoritmos tais como os de difusão tolerantes a falhas. Para suprir tal deficiência, foi proposto neste trabalho o algoritmo *DETECT*, especificamente desenvolvido para realizar o processo de detecção e localização de falhas no hipercubo. O algoritmo *DETECT* se destaca especialmente pela sua eficiência em fornecer informação da configuração de falhas local aos nós, já que se baseia na topologia do hipercubo e tira proveito da mesma. O seu método de funcionamento, bem como os aspectos teóricos que dão base à sua proposição, foram apresentados e demonstrou-se a sua correção. Também foram feitas análises sobre seu desempenho baseadas nos resultados de testes realizados com o mesmo em um hipercubo comercial.

Os algoritmos de difusão de mensagens tolerantes a falhas para hipercubos defeituosos foram analisados em detalhes. As principais características destes algoritmos foram desta-

cadas e apresentadas de forma a facilitar a comparação entre os mesmos. Como fruto deste estudo em particular, foi proposto um novo e eficiente algoritmo chamado *ESLAB*. Este algoritmo possui as principais e mais vantajosas características dos algoritmos propostos até então, entre elas o tratamento de até $n-1$ falhas para um hipercubo n-dimensional e a tolerância a falhas de nó, enlace ou uma combinação de ambos. Demonstrou-se através de lemas, teoremas, exemplos e figuras que o algoritmo *ESLAB* é correto. Além disso, ele foi extensamente testado em um hipercubo comercial, sendo seu desempenho analisado e comparado com o de outros dois algoritmos similares a ele em propósito. Aplicações para *ESLAB* foram também discutidas nesta dissertação, concluindo-se ser este uma boa opção para realização de difusões tolerantes a falhas em hipercubos físicos ou lógicos que apresentam defeitos.

6.2 Contribuições

Dentre as contribuições oferecidas por este trabalho, podemos citar:

1. a análise e comparação sob uma mesma métrica de algoritmos tolerantes a falhas destinados à difusão de mensagens em hipercubos defeituosos; esta análise possibilitou a criação de uma tabela contendo as principais informações sobre estes algoritmos de forma a facilitar a escolha daquele que melhor se enquadra para uma determinada aplicação ou um determinado ambiente;

2. a proposição de um algoritmo simples e eficiente para detecção e localização de falhas em hipercubos;

3. a proposição de um novo algoritmo tolerante a falhas para difusão de mensagens em hipercubos defeituosos que apresenta várias vantagens em relação àqueles já propostos com a mesma finalidade.

6.3 Publicações e Apresentações

Durante o período de pesquisas e estudos para o desenvolvimento deste trabalho, foram feitas as seguintes publicações e apresentações.

Autores:	Saulo R. Nascimento e Marco A. A. Henriques
Título:	An Analysis and Comparison of Algorithms for Broadcasting in Injured Hypercube Networks
Local:	Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, Nevada, USA
Data:	Julho de 1999

Autores:	Saulo R. Nascimento e Marco A. A. Henriques
Título:	An Efficient Algorithm for Broadcasting in Hypercube Networks with Link and Node Failures
Local:	Anais do VIII Simpósio de Computadores Tolerantes a Falhas (SCTF'99), Campinas, SP
Data:	Julho de 1999

Autores:	Saulo R. Nascimento e Marco A. A. Henriques
Título:	Um Algoritmo Distribuído para Detecção e Localização de Falhas em Redes tipo Hipercubo
Local:	Proceedings of the 11^{th} Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'99), Natal, RN
Data:	Outubro de 1999

6.4 Trabalhos futuros

Entre as propostas para trabalhos futuros nesta área, está a de se ampliar a abrangência do algoritmo *DETECT* para que venha a tratar também os casos de falhas transientes que são mais freqüentes e difíceis de tratar que as falhas permanentes. A questão dos dois intervalos de tempo (*timeout*) presentes no *DETECT*, um que os nós esperam antes de considerar um vizinho falho caso não recebam uma mensagem do mesmo, e o outro no qual os nós esperam receber alguma notificação de algum vizinho a respeito de possuírem ou não enlaces falhos, deve ser pesquisada mais profundamente. Se estes intervalos de tempo forem mal definidos, o algoritmo pode não funcionar corretamente.

Possivelmente, uma continuidade nos estudos sobre algoritmos de difusão venha aperfeiçoar o algoritmo *ESLAB* e diminuir o número máximo de etapas que ele exige para completar uma operação de difusão com sucesso. Há um número considerável de trocas

de mensagens neste algoritmo, especialmente devido à comunicação que existe entre os nós perfeitos e não-perfeitos. Um estudo mais aprofundado sobre o relacionamento entre estes dois tipos de nós e sobre o caminho percorrido por uma mensagem durante uma difusão baseada em níveis de segurança, pode resultar em métodos mais eficientes de se detectar que um nó não recebeu uma mensagem sem que haja necessidade de envio de notificações.

Com relação aos testes, seria muito interessante poder executar tanto *DETECT* quanto *ESLAB* em um outro hipercubo que fosse mais flexível que o usado neste trabalho. O n-Cube2 restringe significativamente as formas de comunicação entre os nós. Ele não permite um conhecimento mais detalhado sobre os fatores que influenciam nos tempos de envio e recebimento das mensagens e não tem mecanismos de comunicação mais simples e mais rápidos que os utilizados nos testes. Como apresentado no Apêndice A.1, o tempo gasto numa operação de envio de mensagem, por exemplo, é 200 vezes maior que numa operação de atribuição simples. Além disso as primitivas de comunicação existentes não permitem o envio de mensagens para um subconjunto qualquer de nós do hipercubo (ou subcubo). O n-Cube2 também limita as opções para manipulação dos componentes de um subcubo, não permitindo, por exemplo, que se possa executar um programa em um número de nós qualquer definido pelo usuário.

Resultados mais precisos que os conseguidos neste estudo poderiam ser obtidos a partir de testes realizados em um hipercubo que permitisse simular falhas de nós e enlaces de uma maneira mais próxima possível de um caso real, possivelmente permitindo injeção de falhas por software fazendo com que o kernel do sistema, por exemplo, possa capturar mensagens enviadas para um determinado nó ou por um determinado enlace. Tal hipercubo deveria também possibilitar uma forma mais adequada de se obter os tempos de execução dos algoritmos.

Bibliografia

[ARM 81] J. R. Armstrong and F. G. Gray *Fault Diagnosis in a Boolean n-Cube Array of Microprocessors.* IEEE Transactions on Computers, Vol. C-30, No. 8, Aug. 1981.

[BAN 90] Prithviraj Banerjee, Joe T. Rahmeh, Craig Stunkel, , V. S. Nair, Kaushik Roy, Vilay Balasubramanian and Jacob A. Abraham *Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor.* IEEE Transactions on Computers, Vol. 39, No. 9, Sep. 1990.

[BRU 94] Jehoshua Bruck. *An Optimal Broadcasting in Faulty Hypercubes.* Discrete Applied Mathematics 53, 3–13, 1994.

[CHE 97] Hsing-Lung Chen and Nian-Feng Tzeng *Subcube Determination in Faulty Hypercubes.* IEEE Transactions on Computers, Vol. 46, No. 8, Aug. 1997.

[DEE 99] D. Medhi *Network Reliability and Fault Tolerance.* (invited paper) Wiley Encyclopedia of Electrical and Electronics Engineering, Vol. 14, [Ed. J. G. Webster], 213–218, 1999.

[DIL 84] E. Dilger and E. Ammann *System Level Self-Diagnosis in n-Cube Connected Multiprocessor Networks.* Proc. 14th Int'l Symp. Fault-Tolerant Computing, Kissimmee, FL, 184–189, Jun. 1984.

[DUA 98] E. P. Duarte Jr., T. Nanya, G. Mansfield and S. Nogushi *Non-Broadcast Network Fault-Monitoring Based on System-Level Diagnosis.* Proc. IEEE/IFIP NOMS'98, 597–609, 1998.

[HIL 85] W. D. Hillis *The Connection Machine.* Cambridge, Mass.: MIT Press, 1985.

[HUE 98] Eduardo J. Huerta e Marco A. A. Henriques *Uma Introdução a JOIN: Um Sistema para o Processamento Massivamente Paralelo na Internet.* Relatório Técnico DCA-RT 02/98, FEEC/UNICAMP, Mar. 1998.

[IYE 83] R. K. Iyer and D. J. Rossetti *Permanent CPU Errors and System Activity: Measurement and Modeling.* Proc. Real-Time Syst. Symp., 1983.

[JOH 89] S. L. Johnsson and C. T. Ho. *Optimum broadcasting and personalized communication in hypercubes.* IEEE Transaction on Computers, 1249–1268, 1989.

[JWU 94] Jie Wu. *An Optimal Fault-Tolerant NonRedundant Broadcasting Scheme in Injured Hypercubes.* Journal of Parallel and Distributed Computing, Vol. 22, 295–313, 1994.

[JWU 95] Jie Wu. *Safety-Levels - An Efficient Mechanism for Achieving Reliable Broadcasting in Hypercubes.* IEEE Transactions on Computers, Vol. 44, No. 5, May 1995.

[JWU'95] Jie Wu. *Unicasting in Faulty Hypercubes Using Safety Levels.* IEEE Transactions on Computers, Vol. 46, No. 2, 241–247, Feb. 1995.

[JWU 96] Jie Wu. *Optimal Broadcasting in Hypercubes with Link Faults Using Limited Global Information.* Journal of Systems Architecture 42, 367–380, 1996.

[KUH 81] J. G. Kuhl and S. M. Reddy *Fault Diagnosis in Fully Distributed Systems.* Proc. 11th Int'l Symp. Fault-Tolerant Computing, 100–105, Jun. 1981.

[LEE 92] Tze Chiang Lee and John P. Hayes. *A Fault-Tolerant Communication Scheme for Hypercubes Computers.* IEEE Transactions on Computers, Vol. 41, No. 10, Oct. 1992.

[LEU 96] Yuh-Rong Leu and Sy-Yen Kuo *A fault Tolerant Tree Communication Scheme for Hypercube Systems.* IEEE Transactions on Computers, Vol. 45, No. 6, 641–650, Jun. 1996.

[MAI 99] J. E. B. Maia e M. A. C. Branco *Diagnose de Sistema Considerando Falhas nos Links.* Proc. VIII SCTF, Campinas, Brasil, 54–58, 1999.

[NCU 90] NCUBE Corporation *n-Cube-2 Processor Manual.* NCUBE Corporation, 1990.

[PAR 92] S. Park and B. Bose *Broadcasting in Hypercubes with Link/Node Failures.* Proc. of the 4th Symp. Frontiers of Massively Parallel Computation, 1992.

[RAN 95] Sampath Rangarajan, Anton T. Dahbura and Eric A. Ziegler *A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies.* IEEE Transactions on Computers, Vol. 44, No. 2, Feb. 1995.

[RAT 85] J. Rattner *Concurrent Processing: A New Direction in Scientific Computing.* Proc. of the 1985 National Computer Conference, Vol. 54, 157–166, 1985.

[REE 87] Daniel A. Reed and Richard M. Fujimoto *Multicomputer Networks - Message-Based Parallel Processing.* Scientific Computation Series, MIT Press, 1987.

[REN 84] D. A. Rennels *Fault Tolerant Computing - Concepts and Examples.* IEEE Transactions on Computers, Vol. C-33, 1116–1129, Dec. 1984.

[REN 96] D. A. Rennels *On Implementing Fault-Tolerance in Binary Hypercubes.* Proc. 16th Int'l Symp. Fault-Tolerant Computing, 344–349, Jul. 1996.

[SEI 85] C. L. Seitz *The Cosmic Cube.* Communications of the ACM, Vol. 28, No. 1, 22–33, Jan. 1985.

[SUL 77] H. Sullivan and T. R. Bashkow *A large Scale, Homogeneous, Fully Distributed Parallel Machine.* Proc. of the 4th Annual Symp. Comp. Archit., Vol. 5, No. 7, 105–117, Mar. 1977.

Apêndice A

Aspectos relevantes sobre os testes

A.1 Valores de tempo de operações no n-Cube2

Todos os testes realizados neste trabalho foram feitos em um hipercubo n-Cube2, da empresa NCUBE, de 64 processadores com 4 Mbytes de memória local cada um e clock de 25MHz. Este equipamento possui um sistema operacional próprio e suporta a execução de programas escritos em C e em FORTRAN. Quando se fez a análise teórica de desempenho dos algoritmos implementados neste trabalho, assumiu-se que os valores médios de tempo de execução de operações, tais como as de atribuição, as aritméticas e inclusive as de comunicação, eram aproximadamente os mesmos. Essa suposição decorreu do fato de que esses algoritmos poderiam ser implementados em qualquer máquina e em qualquer linguagem de programação e não se teria como estimar qual realmente seria a relação entre os tempos de execução dessas operações. Uma vez que se tornou possível utilizar o n-Cube2, percebeu-se a importância de se fazer o levantamento dos tempos reais das operações mais comuns utilizadas nos códigos, quando executadas nesta máquina, para que se pudesse avaliar quão perto ou quão longe as estimativas teóricas estavam deste caso real. A seguir são detalhadas as condições específicas dos testes no hipercubo comercial n-Cube2.

Todos os algoritmos testados neste estudo foram codificados em C. O n-Cube2 possui uma biblioteca especial de funções em C para comunicação. As duas funções básicas de comunicação desta biblioteca são a *nread*, para leitura de mensagens que chegam ao buffer de entrada, e a *nwrite* para envio de mensagens. Estas duas em conjunto com a função

ntest, que checa a presença ou não de novas mensagens no buffer de entrada, foram as únicas funções de comunicação utilizadas nos códigos. Para os testes, foram realizadas 10, 100 e 1000 repetições em seqüência das operações de *ntest*, atribuição simples e atribuição de string, e 10, 100 e 500 de *nread* e *nwrite*. Verificou-se que a diferença nos valores medidos com a variação do número de repetições em seqüência era insignificante. Nas operações de *nread*, os nós liam mensagens que haviam sido previamente colocadas nos buffers de entrada, e nas operações de *nwrite* os nós enviavam mensagens de 1 byte. Segundo os manuais do n-Cube2, a operação de *nwrite* é assíncrona, ou seja, ela não trava o processo para esperar confirmação de recebimento da mensagem. Os valores médios dos dados colhidos, e que são apresentados na tabela abaixo, estão coerentes com os dados encontrados em estudos feitos no n-Cube2 por outros pesquisadores do departamento onde esta máquina se encontra instalada.

Tabela A.1: Esforço computacional de algumas operações no n-Cube2

Operação	Tempo (μs)
Atribuição simples	0,73
Atribuição de string	5,25
ntest	59,48
nread	56,75
nwrite	143,91

Como se pode ver, as operações de comunicação no n-Cube2 são muito mais custosas que outras operações, sendo que, por exemplo, o tempo gasto para uma operação de escrita, envio de mensagem, chega a ser quase 200 vezes mais demorada que uma operação de atribuição simples. Em um ambiente como esse, algoritmos que se utilizam de muita comunicação entre os nós têm o seu desempenho prejudicado.

A.2 Confiabilidade dos valores de tempo medidos

Quando testes são realizados, quaisquer que sejam os tipos, sempre se tem a preocupação de se colher os dados resultantes das mesmas da forma mais fiel possível. Especialmente no caso de medição de tempo de execução, esta é uma tarefa bem difícil,

principalmente em um sistema multicomputador. Os dados obtidos podem apresentar interferência de vários fatores. Porém, como os testes realizados neste trabalho não têm caráter de benchmark mas servem apenas para efeito de comparação, a única preocupação que se teve ao se fazer as medições foi a de não se estar favorecendo ou prejudicando nenhum dos algoritmos comparados.

Tanto no caso dos testes com *DETECT* quanto com os algoritmos *ESLAB*, Jie Wu (1995) e Lee & Hayes (1992), o tempo de execução do algoritmo foi sempre considerado como o tempo de execução no nó que registrou o tempo mais longo. Esta é uma aproximação boa já que a contagem do tempo se inicia quase que simultaneamente nos nós. Ela começa no momento em que o programa é carregado nos mesmos, independente do nó que inicia o processo. A Fig. A.1 exemplifica esta questão exibindo o que ocorre nos nós enquanto eles executam o algoritmo *DETECT*. Cada linha na figura indica um nó específico.

De um modo geral, pode-se dividir o tempo que é contado desde o momento em que se

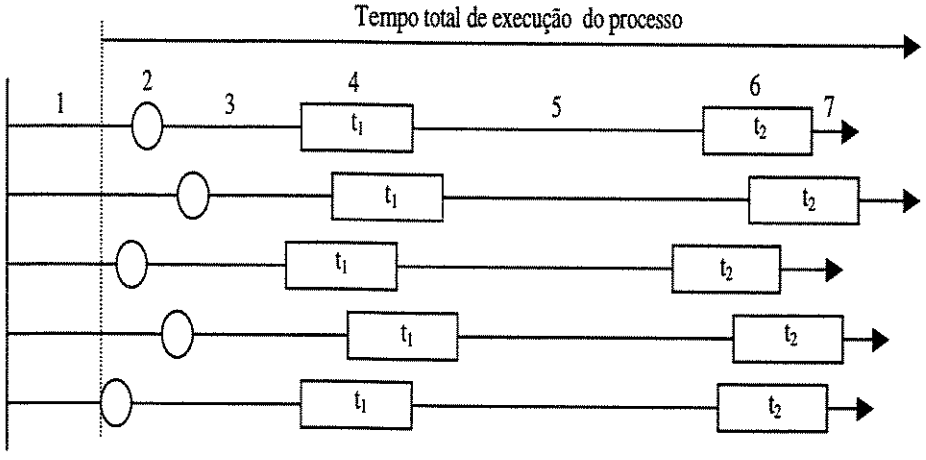

Figura A.1: Medição de tempo no algoritmo *DETECT*

submete o programa para execução até que o último nó termine de executá-lo em 7 partes como mostrado na figura. Estas 7 partes são descritas abaixo.

1. indica o tempo gasto para que o programa seja carregado em cada nó;
2. o nó recebe a mensagem e inicia o programa;
3. as instruções são executadas e as mensagens enviadas aos vizinhos;

4. um tempo t_1 é esperado para garantir que todos os nós não falhos tenham enviado suas mensagens. Na implementação feita neste trabalho, t_1 é função do tamanho do cubo, pois quanto maior o número de dimensões, maior é o número de mensagens que um nó deve enviar, e consequentemente mais tempo ele leva para terminar esta tarefa;

5. outras instruções são realizadas para a localização das falhas. Os nós que detectam enlaces falhos enviam então notificação aos vizinhos não-perfeitos;

6. um tempo t_2 é esperado para garantir que todos os nós que detectaram enlaces falhos tenham enviado as notificações. Na implementação feita neste trabalho, t_2 é função do tamanho do cubo, pois quanto maior o número de dimensões, maior pode ser o número de notificações que um nó deve enviar, e consequentemente mais tempo ele leva para terminar esta tarefa;

7. os nós então verificam se receberam mensagens, lêem-nas (os que possuem enlaces falhos) e atualizam seus vetores de falhas.

Nota-se que o tempo total de execução do processo vai desde o momento em que o nó que inicia o processo entra no intervalo 2 até que o último nó termine o intervalo 7. No exemplo, o nó que inicia o processo é o primeiro a receber o programa, última linha na figura, e o último nó a finalizar as suas tarefas é o representado pela segunda linha na figura. Porém, o tempo medido pelo método adotado é o maior dos 5 intervalos entre a parte de tempo 2 e a 7. Segundo os manuais do n-Cube2, a forma com que é implementada a carga de programas nos nós faz com que a diferença entre as partes de tempo 1 dos nós seja desprezível. Desta forma, o tempo de execução calculado pelo método utilizado se aproxima bastante do tempo total real de execução do algoritmo. A Fig. A.2 mostra um exemplo similar para o caso de execução do algoritmo *ESLAB*.

O tempo neste caso pode ser dividido em 3 partes como descrito abaixo.

1. indica o tempo gasto para que o programa seja carregado em cada nó;

2. o nó recebe a mensagem e inicia a difusão;

3. tempo que cada nó gasta durante a operação.

Neste caso, o tempo total da operação é o tempo decorrido do momento em que o nó fonte inicia o processo até que o último nó receba a mensagem. Através de uma análise

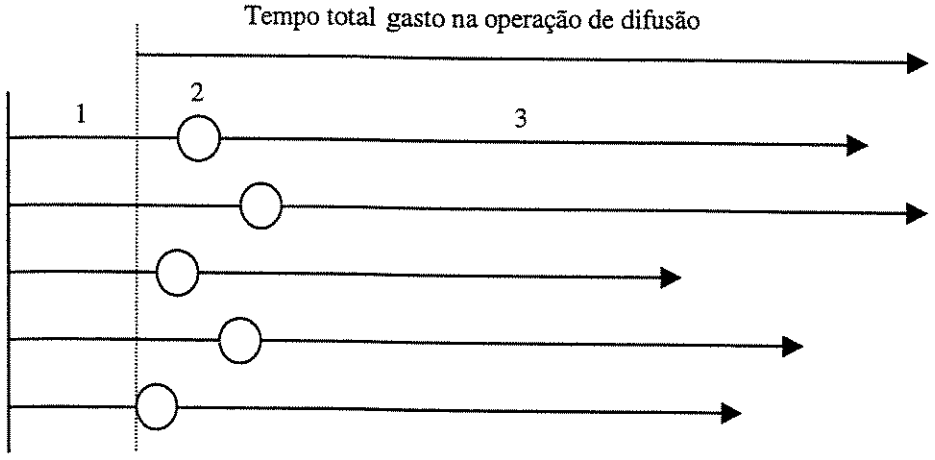

Figura A.2: Medição de tempo no algoritmo *ESLAB*

similar à feita acima, percebe-se que a aproximação do maior tempo medido nos nós do tempo total real da operação é válida. Mesmo que haja alguma discrepância significativa entre esses dois valores, essa diferença dever-se-á ao problema da falta de simultaneidade da carga dos programas nos nós, e este problema será o mesmo para qualquer que seja o algoritmo simulado. Desta forma, o método adotado não estará favorecendo nem prejudicando qualquer algoritmo.

A.3 Configurações de falhas

Nesta dissertação foram utilizados subcubos de 3, 4, 5 e 6 dimensões nos testes feitos. Para ter-se uma noção da quantidade de simulações que deveriam ser realizadas, foi feito um levantamento do número de combinações de falhas possível para cada um desses hipercubos. Antes de mostrar as tabelas com o resultados, será mostrado um exemplo de como os cálculos foram realizados.

Considere a tarefa de encontrar o número total de casos de falhas para um hipercubo de 4 dimensões com 3 falhas onde duas delas são de nó em uma de enlace. Um hipercubo de 4 dimensões possui $2^n = 2^4 = 16$ nós e $n2^{n-1} = 4 \cdot 2^{4-1} = 32$ enlaces. Fixando-se o nó origem de forma a não permitir que ele seja falho, qualquer que seja este nó, têm-se ainda 15 nós como candidatos a serem falhos. Neste caso específico, um par qualquer desses nós

será falho. Tem-se então que:

$$N_f = \begin{pmatrix} 15 \\ 2 \end{pmatrix} = 105 \tag{A.1}$$

onde N_f denota o número de combinações de pares de nós falhos. Dos 32 enlaces do cubo, qualquer um deles pode ser o falho. Porém, deve-se desconsiderar os 4 enlaces de cada um dos 2 nós falhos, pois um enlace falho de um nó falho não conta como mais uma falha. Desta forma tem-se:

$$L_f = 32 - 2 \cdot 4 = 24 \tag{A.2}$$

onde L_f denota o número de enlaces que podem ser falhos neste caso. Note-se que o produto de N_f por L_f não pode ser considerado o número total de combinações de falhas deste exemplo pois não se está considerando o caso dos dois nós falhos serem vizinhos. Quando isto ocorre, temos que $L_f = 25$ e não 24, já que os dois nós falhos compartilham um mesmo enlace. No hipercubo em questão, temos 32 pares de nós vizinhos, que é exatamente o número de enlaces do mesmo. Desses 32 pares, 4 envolvem o nó origem e portanto não devem ser considerados já que o nó origem não pode ser falho. Então, para um hipercubo de 4 dimensões, o número de casos de 3 falhas sendo duas de nó e uma de enlace é:

$$N_f \cdot L_f + (32 - 4) = 2520 + 28 = 2548 \tag{A.3}$$

O cálculo das demais combinações apresentadas nas tabelas a seguir foi realizado seguindo o mesmo raciocínio. Considera-se apenas um máximo de $n - 1$ falhas, para um hipercubo n-dimensional, já que este é o número máximo de falhas tratado neste trabalho. O caso de 0 falhas é também contado como um caso de falha. O motivo para isto é que, ao se calcular o número de casos de testes para um determinado subcubo, deve-se também considerar o caso em que não há falhas no mesmo. Na coluna de tipo de falha, N indica falha de nó e L falha de enlace.

Tabela A.2: Número de combinações de falhas para um hipercubo de 3 dimensões

N^o de falhas	Tipo de falha	Combinações
0		1
1	N	7
	L	12
2	NN	21
	NL	63
	LL	66
Total		**170**

Tabela A.3: Número de combinações de falhas para um hipercubo de 4 dimensões

N^o de falhas	Tipo de falha	Combinações
0		1
1	N	15
	L	32
2	NN	105
	NL	420
	LL	496
3	NNN	455
	NNL	2548
	NLL	5670
	LLL	4960
Total		**14702**

Tabela A.4: Número de combinações de falhas para um hipercubo de 5 dimensões

Nº de falhas	Tipo de falha	Combinações
0		1
1	N	31
	L	80
2	NN	465
	NL	2325
	LL	3160
3	NNN	4495
	NNL	32625
	NLL	86025
	LLL	82160
4	NNNN	31465
	NNNL	294350
	NNLL	1123050
	NLLL	2093275
	LLLL	1581580
Total		**5335087**

Tabela A.5: Número de combinações de falhas para um hipercubo de 6 dimensões

N^o de falhas	Tipo de falha	Combinações
0		1
1	N	63
	L	192
2	NN	1953
	NL	11718
	LL	18336
3	NNN	39711
	NNL	351726
	NLL	1083915
	LLL	1161280
4	NNNN	595665
	NNNL	6921060
	NNLL	31463016
	NLLL	66480120
	LLLL	54870480
5	NNNNN	7028847
	NNNNL	100412100
	NNNLL	597701607
	NNLLL	1866794766
	NLLLL	3041465490
	LLLLL	2063130048
Total		**7839532094**

Apêndice B

Pseudo-Códigos dos Algoritmos Implementados

Os códigos apresentados a seguir são a implementação em C dos algoritmos que foram testados no n-Cube2 assim como descrito nos Cap. 4 e 5. Algumas pequenas omissões e alterações foram feitas nestes códigos para facilitar o seu entendimento, porém as partes mais importantes dos mesmos permaneceram inalteradas.

B.1 Código do algoritmo *DETECT*

```
/* Declaracao de variaveis */
int mynode,myproc,myhost,mask,mydim /* variáveis relativas ao nó e ao cubo */
int nfailures; /* número de falhas de nó */
char fault[6];  /* vetor de falhas de nó */
char enlace[6]; /* vetor de falhas de enlace */
char matriz[6][6]; /* matriz de falhas */

Begin

/* colha informações sobre o nó e o hipercubo */
whoami(&mynode,&myproc,&myhost,&mydim);

nfailures = parâmetro(1);

/* caso o nó esteja na lista dos falhos, não deve proseguir na execução do
   código */
```

```
For (z=0;z<nfailures;z++) {
   If (parâmetro(z+2) = mynode) {
      kill = 1;
      Exit For;
   }
}

/* código para mensagem tese */
type = 10;

If (kill = 0) {

   /* envie mensagem a todos os vizinhos conectados por enlaces não falhos */
   For (i=0; i<mydim; i++) {
      send = 1;
      mask = (1<<i);
      /* gera endereço do vizinho */
      viz = mynode XOR mask;
      fault[i] = 1;
      enlace[i] = 0;
      For (z=0;z<(num_parametros-2-nfailures);z++) {
         If (Compare(parâmetro(z+nfailures+2),(mynode,viz))=0) {
            send = 0;
            Exit For;
         }
      }
      If (send = 1) nwrite(bufw,sizeof(bufw),viz,type,NULL);
   }

   /* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
   Atrasa(x);

   /* testa chegada de mensagem em cada dimensão e atualiza vetor fault */
   For (i=0; i<mydim; i++) {
      mask = (1<<i);
      viz = mynode XOR mask;
      If (ntest(&viz,&type) >= 0) {
         fault[i] = 0;
      }
   }

   /* código relativo ao vetor fault */
   type = 100;

   bufw = fault;

   /* envia o vetor fault a todos os vizinhos do qual recebeu mensagem */
   For (i=0; i<mydim; i++) {
      If (fault[i]=0) {
         mask = (1<<i);
         viz = mynode XOR mask;
         nwrite(bufw,sizeof(bufw),viz,type,NULL);
      }
```

```
  }

  /* recebe o vetor fault dos vizinhos e atualiza matriz de falhas */
  For (i=0; i<mydim; i++) {
     If (fault[i] = 0) {
        mask = (1<<i);
        viz = mynode XOR mask;
        nread(bufr,sizeof(bufr),&viz,&type,NULL);
        matriz[i] = bufr;
     }
  }

  /* codigo relativo a enlace falho */
  type = 50;

  /* verifica se algum vizinho possui enlace falho e faz notificação caso
     descubra */
  For (i=0; i<mydim; i++) {
     If (fault[i] = 0) {
        For (j=0; j<mydim; j++) {
           If ((fault[j]=0) & (i<>j) & (matriz[i][j]=1) & (matriz[j][i]=0)) {
              mask = (1<<i);
              viz = mynode XOR mask;
              bufw = j;
              nwrite(bufw,sizeof(bufw),viz,type,NULL);
           }
        }
     }
  }

  /* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
  Atrasa(y);

  /* verifica chegada de mensagem notificando enlace falho */
  For (i=0; i<mydim; i++) {
     viz = -1; /* qualquer vizinho */
     If (ntest(&viz,&type) >= 0) {
        nread(bufr,sizeof(bufr),&viz,&type,NULL);
        j = bufr;
        enlace[j] = 1;
     }
  }
}

end
```

B.2 Código do algoritmo de Lee & Hayes 92

```
/* Declaração de variáveis */
int mynode,myproc,myhost,mask,mydim /* variáveis relativas ao nó e ao cubo */
int nfailures; /* número de falhas de nó */
```

```
int mysl; /* nível de segurança do nó */
int UNSAFE[6]= { 0, 0, 0, 0, 0, 0}; /* vetor dos vizinhos não-seguros */
int fault[6]; /* vetor dos vizinhos falhos */
int lf; /* quantidade de vizinhos falhos */
int ls; /* quantidade de vizinhos não-seguros */

Begin

kill=0;
nfailures = parâmetro(2);
/* colha informações sobre o nó e o hipercubo */
whoami(&mynode,&myproc,&myhost,&mydim);

/* se está na lista dos falhos, seta variável */
For (z=0;z<nfailures;z++) {
   If (parâmetro(z+3) = mynode) {
      kill = 1;
      Exit For;
   }
}

If (kill = 0) {definestatus();}

type=15;

If (kill = 0) {

   j = parâmetro(1);
   /* se é o nó fonte, então */
   If (mynode = j) {
      For (i=0;i<mydim;i++) {
         dimensões[i] = 1;
      }

      /* se não é ativo, então envia mensagem a um vizinho ativo */
      If (mystatus = 1) {
         For (i=0;i<mydim;i++) {
            mask = (1<<i);
            viz = mynode XOR mask;
            If ((fault[i] = 0) & (UNSAFE[i] = 0)) {
               bufw = dimensões;
               nwrite(bufw,sizeof(bufw),viz,type,NULL);
               i = mydim;
            }
         }
      }

      Else {
         /* difunde para todos vizinhos ativos */
         For (i=0;i<mydim;i++) {
            If (dimensões[i] = 1) {
               If ((fault[i] = 0) & (UNSAFE[i] = 0)) {
                  dimensões[i] = 0;
```

```
                bufw = dimensões;
                mask = (1<<i);
                viz = mynode XOR mask;
                nwrite(bufw,sizeof(bufw),viz,type,NULL);
             }
          }
       }

       /* difunde para todos vizinhos não-seguros */
       For (i=0;i<mydim;i++) {
          If (dimensões[i] = 1) {
             If (UNSAFE[i] = 1) {
                dimensões[i] = 0;
                bufw = dimensões;
                mask = (1<<i);
                viz = mynode XOR mask;
                nwrite(bufw,sizeof(bufw),viz,type,NULL);
             }
          }
       }
    }
}

Else {

   source = -1;
   /* aguarda receber mensagem */
   while (source = -1) {
      For (i=0;i<mydim;i++) {
         mask = (1<<i);
         viz = mynode XOR mask;
         If (ntest(&viz,&type) >= 0) {
            source = i;
            i = mydim;
         }
      }
   }
   nread(bufr,sizeof(bufr),&viz,&type,NULL);
   dimensões = bufr;

   /* difunde aos ativos pelas dimensões atribuídas */
   If (dimensões[i] = 1) {
      If ((fault[i] = 0) & (UNSAFE[i] = 0)) {
         dimensões[i] = 0;
         bufw = dimensões;
         mask = (1<<i);
         viz = mynode XOR mask;
         nwrite(bufw,sizeof(bufw),viz,type,NULL);
      }
   }
}

/* difunde aos não-seguros pelas dimensões atribuídas */
```

```
   For (i=0;i<mydim;i++) {
      If (dimensões[i] = 1) {
         If ((UNSAFE[i] = 1) & (i <> source)) {
            dimensões[i] = 0;
            bufw = dimensões;
            mask = (1<<i);
            viz = mynode XOR mask;
            nwrite(bufw,sizeof(bufw),viz,type,NULL);
         }
      }
   }

}

End

definestatus()

Begin
/* Declaração de variáveis */
int Bn; /* número de trocas de mensagens necessárias */
char enlace[6]; /* vetor de falhas de enlace */
char matriz[6][6]; /* matriz de falhas */

/* código para mensagem tese */
type = 10;

For (i=0; i<mydim; i++) {
   send = 1;
   mask = (1<<i);
   viz = mynode XOR mask;
   fault[i] = 1;
   For (z=0;z<(num_parametros-2-nfailures);z++) {
      If (Compare(parâmetro(z+nfailures+2),(mynode,viz))=0) {
         send = 0;
         Exit For;
      }
   }
   If (send = 1) nwrite(bufw,sizeof(bufw),viz,type,NULL);
}

/* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
Atrasa(x);

/* testa chegada de mensagem em cada dimensão e atualiza vetor fault */
For (i=0; i<mydim; i++) {
   mask = (1<<i);
   viz = mynode XOR mask;
   If (ntest(&viz,&type) >= 0) {
      fault[i] = 0;
   }
}
```

```
/* código relativo ao vetor fault */
type = 100;

bufw = fault;

/* envia o vetor fault a todos os vizinhos do qual recebeu mensagem */
For (i=0; i<mydim; i++) {
   If (fault[i] = 0) {
      mask = (1<<i);
      viz = mynode XOR mask;
      nwrite(bufw,sizeof(bufw),viz,type,NULL);
   }
}
/* recebe o vetor fault dos vizinhos e atualiza matriz de falhas */
For (i=0; i<mydim; i++) {
   If (fault[i] = 0) {
      mask = (1<<i);
      viz = mynode XOR mask;
      nread(bufr,sizeof(bufr),&viz,&type,NULL);
      matriz[i] = bufr;
   }
}

/* código relativo a enlace falho */
type = 50;

/* verifica se algum vizinho possui enlace falho e faz notificação caso
   descubra */
For (i=0; i<mydim; i++) {
   If (fault[i] = 0) {
      For (j=0; j<mydim; j++) {
         If ((fault[j]=0) & (i<>j) & (matriz[i][j]=1) & (matriz[j][i]=0)) {
            mask = (1<<i);
            viz = mynode XOR mask;
            bufw = j;
            nwrite(bufw,sizeof(bufw),viz,type,NULL);
         }
      }
   }
}

/* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
Atrasa(x);

/* verifica chegada de mensagem notificando enlace falho */
For (i=0; i<mydim; i++) {
   viz = -1; /* qualquer vizinho */
   If (ntest(&viz,&type) >= 0) {
      nread(bufr,sizeof(bufr),&viz,&type,NULL);
      j = bufr;
      enlace[j] = 1;
   }
}
```

```
f = ls = mystatus = 0;

/* se possui enlace falho, seta status para não-seguro */
For (i=0; i<mydim; i++) {
   If (enlace[i] = 1) {
      mystatus=1;
    Exit For;
   }
   Else {
      If (fault[i] = 1) {lf++;}
   }
}

/* define número de iterações */
If (Resto(mydim/2) = 0) {Bn = (mydim*mydim/4) + mydim - 9/4;}
Else {Bn=(mydim*mydim/4) + mydim/2 - 1;}

/* código relativo ao status */
type = 35;

/* envia o status aos vizinhos */
For (i=0; i<mydim; i++) {
   mask = (1<<i);
   viz = mynode XOR mask;
   bufw = mystatus;
   If (fault[i] = 0) {nwrite(bufw,sizeof(bufw),viz,type,NULL);}
}

For (j=0; j<Bn; j++) {

   If (mystatus = 0) {

      /* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
      Atrasa(y);

      /* recebe status dos vizinhos e atualiza variáveis */
      For (i=0; i<mydim; i++) {
         mask = (1<<i);
         viz = mynode XOR mask;
         If (ntest(&viz,&type) >= 0) {
            nread(bufr,sizeof(bufr),&viz,&type,NULL);
            If (bufr = 1) {
               UNSAFE[i]=1;
               ls++;
            }
         }
      }

      If ((ls+lf)>i) {
         mystatus = 1;
         /* envia o status aos vizinhos */
         For (i=0; i<mydim; i++) {
```

```
            mask = (1<<i);
            viz = mynode XOR mask;
            bufw = mystatus;
            If (fault[i] = 0) {nwrite(bufw,sizeof(bufw),viz,type,NULL);}
         }
      }
   }

   Else {

      /* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
      Atrasa(p);

      /* recebe status dos vizinhos e atualiza variáveis */
      For (i=0; i<mydim; i++) {
         mask = (1<<i);
         viz = mynode XOR mask;
         If (ntest(&viz,&type) >= 0) {
            nread(bufr,sizeof(bufr),&viz,&type,NULL);
            If (bufr = 1) {UNSAFE[i] = 1;}
         }
      }
   }
}

End
```

B.3 Código do algoritmo de Jie Wu 95

```
/* Declaração de variáveis */
int mynode,myproc,myhost,mask,mydim /* variáveis relativas ao nó e ao cubo */
int nfalhas; /* número de falhas de nó */
int mysl; /* nível de segurança do nó */
int vizsl[6]; /* nível de segurança dos nos vizinhos */
char falhas[6];  /* vetor de falhas de nó */

Begin

nfalhas = parâmetro(2);
For (i=0;i<nfalhas;i++) {
   falhas[i] = parâmetro(i+3);
}

/* define o nível de segurança do no */
definesl();

/* colha informações sobre o nó e o hipercubo */
whoami(&mynode,&myproc,&myhost,&mydim);

type=10;
```

```
/* se não é falho, então */
If (mysl <> 0) {
   j = parâmetro(1);
   /* se é o nó fonte, então */
   If (mynode = j) {
      /* cria vetor de dimensões */
      For (i=0;i<mydim;i++) {
         dimensões[i] = i;
         size_d++;
      }

      /* se não é um nó seguro, então envia a mensagem para um vizinho
         seguro */
      If (mysl<mydim) {
         For (i=0;i<mydim;i++) {
            If (vizsl[i] = mydim) {
               bufw = dimensões;
               mask = (1<<i);
               viz = mynode XOR mask;
               nwrite(bufw,sizeof(bufw),viz,type,NULL);
               /* aguarda incumbência de difusão */
               source = -1;
               while (source = -1) {
                  If (ntest(&viz,&type) >= 0) {
                     source = viz;
                  }
               }
               nread(bufr,sizeof(bufr),&viz,&type,NULL);
               dimensões = bufr;
               size_d = strlen(dimensões);
               resta = size_d;

               /* difunde pelas dimensões atribuídas */
               For (j=0;j<resta;j++) {
                  select_dimension();
                  withdraw(dim);
                  bufw = dimensões;
                  mask = (1<<dim);
                  viz = mynode XOR mask;
                  If (vizsl[dim]>0) {
                     nwrite(bufw,sizeof(bufw),viz,type,NULL);
                  }
               }
               Exit For
            }
         }
      }
      Else {
         /* difunde em todas dimensões possíveis */
         For (i=0;i<mydim;i++) {
            select_dimension();
            withdraw(dim);
            bufw = dimensões;
```

```
            mask = (1<<dim);
            viz = mynode XOR mask;
            If (vizsl[dim] > 0) {
               nwrite(bufw,sizeof(bufw),viz,type,NULL);
            }
         }
      }
   }

   Else {
      source = -1;
      /* aguarda receber mensagem */
      while (source = -1) {
         For (i=0;i<mydim;i++) {
            mask = (1<<i);
            viz = mynode XOR mask;
            If (ntest(&viz,&type) >= 0) {
               source = i;
               Exit For;
            }
         }
      }
      nread(bufr,sizeof(bufr),&viz,&type,NULL);
      dimensões = bufr;
      size_d = strlen(dimensões);
      resta = size_d;

      /* difunde pelas dimensões atribuídas */
      For (i=0;i<resta;i++) {
         select_dimension();
         withdraw(dim);
         bufw = dimensões;
         mask = (1<<dim);
         viz = mynode XOR mask;
         If (dim = source) {
            /* para o nó fonte original envia apenas seqüência de dimensões */
            nwrite(bufw,sizeof(bufw),viz,type,NULL);
         }
         Else {
            If (vizsl[dim] > 0) {
               nwrite(bufw,sizeof(bufw),viz,type,NULL);
            }
         }
      }
   }
}

End

withdraw(int d)

Begin
```

```
int i,j;
   For (i=0;i<size_d;i++) {
      If ((dimensões[i]) = d) {
         For (j=i;j<size_d;j++) {
            dimensões[j] = dimensões[j+1];
               Exit For;
         }
      }
   }
   size_d--;

End

select_dimension()

Begin

int bigger,i,j;
   bigger=0;
   dim = dimensões[0];
   For (i=0;i<size_d;i++) {
      j = dimensões[i];
         If (vizsl[j] > bigger) {
            bigger = vizsl[j];
            dim = j;
         }
   }

End

definesl()

Begin

/* Declaração de variáveis */
int falhos[7] = { 0, 0, 0, 0, 0, 0, 0};
int levels[7] = { 0, 1, 2, 3, 4, 5, 6};

kill=0;

/* se está na lista dos falhos, seta variável */
For (z=0;z<nfalhas;z++) {
   i = falhas[z];
   If (i = mynode) {
      kill = 1;
      z = nfalhas;
   }
}

If (kill = 0) {

   int vizsl_[7] = { 0, 0, 0, 0, 0, 0, 0};
   mysl = mydim;
```

```
/* envia o nível de segurança aos vizinhos */
For (i=0; i<mydim; i++) {
   mask = (1<<i);
   viz = mynode XOR mask;
   bufw = mysl;
   nwrite(bufw,sizeof(bufw),viz,type,NULL);
}

/* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
Atrasa(x);

/* recebe níveis de segurança dos vizinhos */
For (i=0; i<mydim; i++) {
   mask = (1<<i);
   viz = mynode XOR mask;
   If (ntest(&viz,&type) >= 0) {
      nread(bufr,sizeof(bufr),&viz,&type,NULL);
      j = bufr;
      vizsl_[i] = j;
      falhos[i] = 1;
   }
}

/* ordena níveis de segurança */
qsort(vizsl_, mydim, sizeof(int), comp);
/* define o nível de segurança */
For (i=0; i<mydim; i++) {
   If (vizsl_[i] < levels[i]) {
      mysl = i;
      Exit For;
   }
}

/* repete processo até estabilizar os níveis de segurança */
For (z=1; z<mydim; z++) {
   int vizsl_[7] = { 0, 0, 0, 0, 0, 0, 0};
   For (i=0; i<mydim; i++) {
   mask = (1<<i);
   viz = mynode XOR mask;
   bufw = mysl;
   nwrite(bufw,sizeof(bufw),viz,type,NULL);
}

For (i=0; i<mydim; i++) {
   mask = (1<<i);
   viz = mynode ^ mask;
   If (falhos[i] = 1) {
      nread(bufr,sizeof(bufr),&viz,&type,NULL);
      j = bufr;
      vizsl_[i] = j;
   }
}
```

```
   If (z < mydim-1) {
      qsort(vizsl_, mydim, sizeof(int), comp);
      For (i=0; i<mydim; i++) {
         If (vizsl_[i] < levels[i]) {
            mysl = i;
            i = mydim;
         }
      }
   }
   Else {
      For (i=0; i<mydim; i++) {vizsl[i] = vizsl_[i];}
   }
}

End

comp(const int *i, const int *j)
Begin
 return *i-*j;
End
```

B.4 Código do algoritmo *ESLAB*

```
/* Declaração de variáveis */
int mynode,myproc,myhost,mask,mydim /* variáveis relativas ao nó e ao cubo */
int nfailures; /* número de falhas de nó */
int mysl; /* nível de segurança do nó */
int vizsl[6]; /* nível de segurança dos nos vizinhos */
char falhas[6]; /* vetor de falhas de nó */
int haveflink; /* indica se o no e não-perfeito */
int fl_list[7]; /* vetor dos vizinhos não-perfeitos */
int ak_list[7]; /* vetor dos vizinhos perfeitos */

Begin

nfalhas = parâmetro(2);

/* colha informações sobre o nó e o hipercubo */
whoami(&mynode,&myproc,&myhost,&mydim);
mysl=mydim;

For (z=0;z<nfailures;z++) {
   If (parâmetro(i+3) = mynode) {
      mysl = 0;
      Exit For;
   }
}

If (mysl <> 0) {definesl();}

/* código para mensagem de difusão */
```

```
type = 19;
difunde = 1

/* se não é falho, então */
If (mysl <> 0) {
   j = parâmetro(1);
   /* se é o nó fonte, então */
   If (mynode = j) {
      /* cria vetor de dimensões */
      For (i=0;i<mydim;i++) {
         dimensões[i] = i;
         size_d++;
      }

      /* se não é um nó seguro, então envia a mensagem para um vizinho
         seguro */
      If (mysl<mydim) {
         For (i=0;i<mydim;i++) {
            If (vizsl[i] = mydim) {
               bufw = dimensões;
               mask = (1<<i);
               viz = mynode XOR mask;
               nwrite(bufw,sizeof(bufw),viz,type,NULL);
               Exit For;
            }
         }
      }

      Else {
         difunde=0;
         /* difunde em todas dimensões possíveis */
         For (i=0;i<mydim;i++) {
            select_dimension();
            withdraw(dim);
            bufw = dimensões;
            mask = (1<<dim);
            viz = mynode XOR mask;
            If (vizsl[dim] > 0) {
               nwrite(bufw,sizeof(bufw),viz,type,NULL);
            }
         }
      }

   }

   If (difunde = 1) {
      source = -1;
      /* aguarda receber mensagem */
      while (source = -1) {
         For (i=0;i<mydim;i++) {
            mask = (1<<i);
            viz = mynode XOR mask;
            If (ntest(&viz,&type) >= 0) {
```

```
            source = i;
            Exit For;
         }
      }
   }
   nread(bufr,sizeof(bufr),&viz,&type,NULL);
   dimensões = bufr;
   size_d = strlen(dimensões);
   resta = size_d;

   /* se é não-perfeito, então */
   If (haveflink = 1) {
      ak_list[source] = 0;
      /* difunde pelas dimensões atribuídas */
      For (i=0;i<resta;i++) {
         select_dimension();
         withdraw(dim);
         ak_list[dim] = 0;
         bufw = dimensões;
         mask = (1<<dim);
         viz = mynode XOR mask;
         If (dim = source) {
            /* para o nó fonte original envia apenas seqüência de
               dimensões */
            nwrite(bufw,sizeof(bufw),viz,type,NULL);
         }
         Else {
            If (vizsl[dim]>0) {
               nwrite(bufw,sizeof(bufw),viz,type,NULL);
            }
         }
      }

      /* notifica vizinhos perfeitos sobre recebimento da mensagem */
      j = sizeof(ak_list);
      If (j>0) {
         For (i=0;i<mydim;i++) {
            /* código para notificação */
            type = 21;
            If (ak_list[i] = 1) {
               mask = (i<<i);
               viz = mynode XOR mask;
               nwrite(bufw,sizeof(bufw),viz,type,NULL);
            }
         }
      }
   }

   /* se é perfeito, então */
   Else {
      fl_list[source] = 0;
      /* difunde pelas dimensões atribuídas */
      For (i=0;i<resta;i++) {
```

```
   select_dimension();
   withdraw(dim);
   fl_list[dim] = 0;
   bufw = dimensões;
   mask = (1<<dim);
   viz = mynode XOR mask;
   If (dim = source) {
      /* para o nó fonte original envia apenas seqüência de
         dimensões */
      nwrite(bufw,sizeof(bufw),viz,type,NULL);
   }
   Else {
      If (vizsl[dim]>0) {
         nwrite(bufw,sizeof(bufw),viz,type,NULL);
      }
   }
}

/* recebe notificações dos vizinhos não-perfeitos */
j = sizeof(fl_list);
If (j > 0) {
   /* a função Atrasa(x) suspende o processo pelo tempo indicado por
      x */
   Atrasa(x);
   For (i=0;i<mydim;i++) {
      mask = (1<<i);
      viz = mynode XOR mask;
      /* código para notificação */
      type = 21;
      If (ntest(&viz,&type) >= 0) {
         fl_list[i] = 0;
      }
   }
}

/* envia a mensagem a um vizinho não-perfeito se for necessário */
j = sizeof(fl_list);
If (j > 0) {
   For (i=0;i<mydim;i++) {
      If (fl_list[i] = i) {
         For (z=0;z<=i;z++) {
            If ((matriz[i][z] = 0) & (z < i)) {z = i+1;}
            Else {
               If (z = i) {
                  mask = (1<<i);
                  viz = mynode XOR mask;
                  type = 19;
                  nwrite(bufw,sizeof(bufw),viz,type,NULL);
               }
            }
         }
      }
   }
```

```
        }
}
   }
}

End

withdraw(int d)

Begin

int i,j;
   For (i=0;i<size_d;i++) {
      If ((dimensões[i]) = d) {
         For (j=i;j<size_d;j++) {
            dimensões[j] = dimensões[j+1];
            Exit For;
         }
      }
   }
   size_d--;

End

select_dimension()

Begin

int bigger,i,j;

   bigger = 0;
   dim = dimensões[0];
   For (i=0;i<size_d;i++) {
      j = dimensões[i];
      If (vizsl[j] > bigger) {
         bigger = vizsl[j];
         dim = j;
      }
   }

End

definesl()

Begin

/* Declaração de variáveis */
int levels[7] = { 0, 1, 2, 3, 4, 5, 6}
char enlace[6]; /* vetor de falhas de enlace */
char matriz[6][6]; /* matriz de falhas */
int vizsl_[7] = { 0, 0, 0, 0, 0, 0, 0};

/* código para mensagem tese */
```

```
type = 10;

For (i=0; i<mydim; i++) {
   send = 1;
   mask = (1<<i);
   viz = mynode XOR mask;
   fault[i] = 1;
   For (z=0;z<(num_parametros-2-nfailures);z++) {
      If (Compare(parâmetro(z+nfailures+2),(mynode,viz))=0) {
         send = 0;
         Exit For;
      }
   }
   If (send = 1) nwrite(bufw,sizeof(bufw),viz,type,NULL);
}

/* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
Atrasa(x);

/* testa chegada de mensagem em cada dimensão e atualiza vetor fault */
For (i=0; i<mydim; i++) {
   mask = (1<<i);
   viz = mynode XOR mask;
   If (ntest(&viz,&type) >= 0) {
      fault[i] = 0;
   }
}

/* código relativo ao vetor fault */
type = 100;

bufw = fault;

/* envia o vetor fault a todos os vizinhos do qual recebeu mensagem */
For (i=0; i<mydim; i++) {
   If (fault[i] = 0) {
      mask = (1<<i);
      viz = mynode XOR mask;
      nwrite(bufw,sizeof(bufw),viz,type,NULL);
   }
}

/* recebe o vetor fault dos vizinhos e atualiza matriz de falhas */
For (i=0; i<mydim; i++) {
   If (fault[i] = 0) {
      mask = (1<<i);
      viz = mynode XOR mask;
      nread(bufr,sizeof(bufr),&viz,&type,NULL);
      matriz[i] = bufr;
   }
}

/* código relativo a enlace falho */
```

```
type = 50;

/* verifica se algum vizinho possui enlace falho e faz notificação caso
   descubra */
For (i=0; i<mydim; i++) {
   If (fault[i] = 0) {
      For (j=0; j<mydim; j++) {
         If ((fault[j]=0) & (i<>j) & (matriz[i][j]=1) & (matriz[j][i]=0)) {
            mask = (1<<i);
            viz = mynode XOR mask;
            bufw = j;
            nwrite(bufw,sizeof(bufw),viz,type,NULL);
         }
      }
   }
}

/* a função Atrasa(x) suspende o processo pelo tempo indicado por x */
Atrasa(x);

/* verifica chegada de mensagem notificando enlace falho */
For (i=0; i<mydim; i++) {
   viz = -1; /* qualquer vizinho */
   If (ntest(&viz,&type) >= 0) {
      nread(bufr,sizeof(bufr),&viz,&type,NULL);
      j = bufr;
      enlace[j] = 1;
      haveflink=1;
   }
}

/* código relativo ao nível de segurança */
type=15;

j = 0;
For (i=0; i<mydim; i++) {
   If (fault[i] <> 0) {j++;}
}

/* se mais de 2 vizinhos falhos, então nível de segurança é 1 */
If (j >= 2) {mysl = 1;}

/* envia o nível de segurança aos vizinhos */
For (i=0; i<mydim; i++) {
   If (fault[i] = 0) {
      mask = (1<<i);
      viz = mynode XOR mask;
      fault[mydim]= mysl;
      bufw = fault;
      bufw[mydim+1] = sizeof(enlace);
      nwrite(bufw,sizeof(bufw),viz,type,NULL);
   }
}
```

```
/* recebe os níveis de segurança dos vizinhos e atualiza variáveis */
For (i=0; i<mydim; i++) {
   fl_list[i] = 0;
   ak_list[i] = 1;
   If (fault[i] = 0) {
      mask = (1<<i);
      viz = mynode XOR mask;
      nread(bufr,sizeof(bufr),&viz,&type,NULL);
      j = bufr[mydim];
      vizsl_[i] = j;
      If (bufr[mydim+1]>0) {
         If (haveflink = 1){
            ak_list[i] = 0;
            fault[i] = 2;
         }
         Else {
            fl_list[i] = 1;
            fault[i] = 2;
         }
      }
   }
   Else {ak_list[i] = 0;}
}

/* ordena niveis de segurança */
qsort(vizsl_, mydim, sizeof(int), comp);

/* define o nível de segurança */
For (i=0; i<mydim; i++) {
   If (vizsl_[i] < levels[i]) {
      mysl = i;
      i= mydim;
   }
}

/* repete o processo até estabilizar os níveis de segurança */
For (z=1; z<(mydim-1); z++) {
   int vizsl_[7] = { 0, 0, 0, 0, 0, 0, 0};
   For (i=0; i<mydim; i++) {
      If ((fault[i] = 0) OR (fault[i] = 2)) {
         mask = (1<<i);
         viz = mynode XOR mask;
         bufw = mysl;
         If (z = 1) {
            fault[mydim] = mysl;
            bufw = fault;
         }
         nwrite(bufw,sizeof(bufw),viz,type,NULL);
      }
   }

   For (i=0; i<mydim; i++) {
```

```
      If ((fault[i] = 0) OR (fault[i] = 2)) {
         mask = (1<<i);
         viz = mynode XOR mask;
         nread(bufr,sizeof(bufr),&viz,&type,NULL);
         j = bufr ;
         If (z = 1) {
            j = bufr[mydim];
            matriz[i] = bufr;
         }
         vizsl_[i] = j;
      }
   }

   If (z < mydim-2) {
      qsort(vizsl_, mydim, sizeof(int), comp);
      For (i=0; i<mydim; i++) {
         If (vizsl_[i] < levels[i]) {
            mysl = i;
            i = mydim;
         }
      }
   }
   Else {
      For (i=0; i<mydim; i++) {vizsl[i] = vizsl_[i];}
   }
}

End

comp(const int *i, const int *j)
Begin
 return *i-*j;
End
```