LUIZ CARLOS PESSOA ALBINI

UM ALGORITMO BASEADO EM COMPARAÇÕES PARA DIAGNÓSTICO DISTRIBUÍDO HIERÁRQUICO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Mestrado em Informática. Departamento de Informática. Setor de Ciências Exatas. Universidade Federal do *Paraná*.

Orientador: Prof. Elias P. Duarte Jr.

CURITIBA
2000

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Luiz Carlos Pessoa Albini, avaliamos o trabalho intitulado "**Um Algoritmo Baseado em Comparações para Diagnóstico Distribuído Hierárquico**", cuja defesa foi realizada no dia 25 de agosto de 2000. Após a Avaliação, decidimos pela Aprovação do Candidato.

Curitiba, 25 de agosto de 2000.

Prof. Dr. Elias Procópio Duarte Jr.
Presidente - DINF/UFPR

Prof. Dr. Paolo Santi
U.PISA/ITÁLIA

Prof. Dr. Martin Alejandro Musicante
DINF/UFPR

Agradecimentos

Aos meus pais pelo seu carinho, apoio e suporte em todos os momentos difíceis de minha vida, e por estarem sempre ao meu lado para compartilhar os momentos alegres e felizes.

A minha avó Olga pelo seu carinho e paciência durante toda a minha estada em Curitiba.

Ao meu orientador prof. Elias Procópio Duarte Jr. por ter me mostrado o caminho e o fascinio pela pesquisa científica, e por ter me apoiado e incentivado durante todo o nosso trabalho.

Aos amigos Alessandro e Rodrigo por todo o apoio, incentivo, sugestões e trabalho mutuo realizado.

Ao amigo Aldri que mostrou-me os primeiros passos para escrever em LaTeX, por ter me emprestado sua dissertação como referência e por todo o seu suporte e incentivo.

Ao amigo Luiz Bona pelas suas correções no texto desta dissertacão.

Aos professores Martin Mussicante e Paolo Santi por terem aceitado fazer parte da banca desta dissertação em tão curto espaço de tempo.

A prof. Cristina Murta por ter possibilitado a vinda do doutor Paolo Santi para a banca e por ter arranjado todos os detalhes para a realização desta banca em tão pouco tempo.

A todos que confiaram e acreditaram em mim.

Conteúdo

Agradecimentos i

Resumo viii

Abstract ix

1 Introdução 1

1.1 Classificação de Falhas 3

1.2 Modelos de Diagnóstico 4

1.3 *Hierarchical Comparison-based Adaptive Distributed System-Level Diagnosis Algorithm* 6

2 Modelos de Diagnóstico em Nível de Sistema 9

2.1 Modelo PMC 10

2.1.1 *Adaptive-DSD* 16

2.1.2 *Hi-ADSD* 19

2.1.3 *Hi-ADSD with Detours* 22

2.1.4 *Hi-ADSD with Timestamps* 24

2.2 Modelos de Diagnóstico Baseados em Comparações 27

2.2.1 Modelo Baseado em Comparações 27

2.2.2 Modelo MM 29

2.2.3 Modelo de Comparações Generalizado 32

2.2.4 Modelo *Broadcast Comparison* 33

3 Um Algoritmo Hierárquico para Diagnóstico Distribuído baseado em Comparações 36

3.1 Modelo de Diagnóstico 36

3.2 O Algoritmo *Hi-Comp* 38

3.2.1 Descrição do Algoritmo 44

3.2.2 Especificação do Algoritmo 47

3.3 Provas 56

4 Resultados de Simulação 62

4.1 Funcionamento do Algoritmo 63

4.1.1 $N-1$ Nodos Sem-Falha 64

4.1.2 1 Nodo Sem-Falha 65

4.2 Latência do Algoritmo 69

4.2.1 Diagnóstico de um Evento 69

4.2.2 Um Nodo Realiza o Diagnóstico de $N-1$ Eventos . . . 71

4.3 Quantidade Máxima de Testes Necessários 73

4.4 Diagnosticabilidade (*Diagnosability*) 76

5 Conclusão **80**

A Programa de Simulação **87**

Lista de Figuras

1.1 Classificação de Falhas. 4

2.1 Grafo do Sistema. 11

2.2 Grafo de Testes. 12

2.3 Síndrome de um sistema de 5 nodos. 13

2.4 Sistema executando o algoritmo *Adaptive-DSD*. 18

2.5 Divisão de clusters do algoritmo *Hi-ADSD*. 20

2.6 Sistema executando o algoritmo *Hi-ADSD*. 21

2.7 Sistema executando o algoritmo *Hi-ADSD with Detours*. . . . 22

2.8 Caminho original e um *detour* entre o nodo 0 e o nodo 7. . . 24

2.9 Divisão de clusters no *Hi-ADSD with Timestamps*. 26

2.10 Grafo do modelo MM 31

3.1 Multi-grafo T(S). 40

3.2 Grafo TFF. 41

3.3 TFF para um sistema de 8 nodos. 41

3.4 TFF_0 para um sistema de 8 nodos. 43

3.5 Um exemplo de $c_{i,s,p}$: $c_{0,3,2}$. 44

3.6 Clusters de um sistema de 8 nodos. 45

3.7 Ilustração de TFF_a. 58

3.8 Caso impossível em um sistema executando o algoritmo *Hi-Comp*. 61

4.1 Sistema com 16 nodos com 1 evento. 64

4.2 Sistema de 16 nodos com 1 evento e somente um nodo sem-falha. 66

4.3 Rodadas de testes necesárias para todos os nodos sem-falha realizem o diagnóstico de 1 evento. 71

4.4 Sistema com 16 nodos com $N - 1$ eventos. 72

4.5 Número de Testes excutado no Sistema. 74

4.6 Pior caso do número de testes para 16 nodos. 75

Lista de Tabelas

4.1 Quantidade de nodos que realizam o diagnóstico de um evento por rodada de testes, para um sistema de 16 nodos. 70

4.2 Quantidade de eventos que o nodo 0 consegue diagnosticar por rodada de testes, para um sistema de 16 nodos. 73

Resumo

Neste trabalho é apresentado o algoritmo *Hi-Comp* (*Hierarchical Comparison-based Adaptive Distributed System-Level Diagnosis algorithm*). Esse algoritmo realiza o diagnóstico de sistemas representáveis por um grafo completo. O algoritmo *Hi-Comp* é o primeiro algoritmo de diagnóstico ao mesmo tempo hierárquico, distribuído e baseado em comparações. Graças à execução de testes através de comparações, o algoritmo não se limita ao diagnóstico de falhas *crash*. Para executar um teste, um processador envia uma tarefa para dois processadores do sistema que, após a executarem, devolvem seus respectivos resultados para o testador. O testador compara os dois resultados; se eles forem idênticos, o testador considera que os dois processadores em questão estão sem-falha; já se a comparação indicar uma divergência entre os dois resultados, o testador considera que pelo menos um dos dois processadores testados está falho, mas não sabe qual deles. Considerando um sistema com N processadores, prova-se que o algoritmo é $N-1$-diagnosticável e possui latência de log_2N rodadas de testes. Além disso, é apresentada a prova formal do número máximo de testes necessários por rodada de testes, que pode chegar a $O(N^3)$ no pior caso. Resultados de simulação são também apresentados.

Abstract

This work introduces a new system-level diagnosis algorithm: *Hi-Comp* (*Hierarchical Comparison-based Adaptive Distributed System-Level Diagnosis algorithm*). This algorithm allows diagnosis of systems that can be represented by a complete graph. *Hi-Comp* is the first diagnosis algorithm that is, at the same time, hierarchical, distributed and comparison-based. The algorithm is not limited to crash fault diagnosis, because its tests are based on comparisons. To perfom a test, a processor sends a task to two processors of the system which, after executing the task, send their outputs to the tester. The tester compares the two outputs; if the comparison produces a match, the tester considers the two processors fault-free; on the other hand, if the comparison produces a mismatch, the tester considers that at least one of the two tested processors is faulty, but can not determine which one. Considering a system of N nodes, it is proved that the algorithm is $(N-1)$-diagnosable and that its latency is $log_2 N$ testing rounds. Furthermore, a formal proof of the maximum number of tests required per testing round is presented, which can be $O(N^3)$. Simulation results are also presented.

Capítulo 1

Introdução

A necessidade de sistemas de computação confiáveis tem motivado pesquisadores a projetar sistemas multiprocessadores auto-diagnosticáveis, ou seja, sistemas compostos por diversos processadores que são capazes de detectar e identificar suas próprias falhas. Identificar falhas nos processadores do sistema, isto é, descobrir quais processadores estão falhos e quais estão sem-falha, é a meta de *diagnóstico em nível de sistema.*

Sistemas multiprocessadores começaram a ser construídos devido à demanda dos aplicativos por sistemas mais poderosos. Existem dois tipos básicos de sistemas multiprocessadores: os sistemas de processamento paralelo e os sistemas distribuídos. Sistemas de processamento paralelo são compostos por diversos processadores que utilizam-se de memória compartilhada para se comunicam. Já sistemas distribuídos são compostos por diversas

unidades independentes, cada unidade possui seu próprio processador e sua própria memória, e elas utilizam-se de trocas de mensagens para se comunicarem. Normalmente essas mensagens são trocadas através de uma rede de comunicação.

Nos sistemas distribuídos as diversas unidades trabalham em conjunto para executar uma determinada tarefa. Se uma unidade do sistema falhar é importante que o sistema como um todo continue funcionando. Para que isso aconteça, quando uma unidade deixar de funcionar, as outras devem passar a realizar a tarefa que não está mais sendo realizada, para tanto faz-se necessário que as unidades do sistema conheçam o estado atual umas das outras; é aqui que entra a tarefa de diagnóstico em nível de sistema: possibilitar que uma unidade que esteja funcionando corretamente saiba do estado atual das outras através da execusão de testes simples.

Cada unidade de um sistema pode estar somente em dois estados, falha ou sem-falha. Porém unidades falhas podem ter comportamentos aleatórios, dependendo do tipo de falha, descritos na próxima seção, que a unidade sofreu.

1.1 Classificação de Falhas

As falhas em sistemas distribuídos podem ser classificadas conforme o comportamento das unidades quando falhas. Uma classificação é apresentada em [16], na qual, as falhas são classificadas em quatro tipos: *Crash Fault, Omission Fault, Timing Fault* e *Byzantine Fault.*

Crash Fault. Falhas crash fazem que a unidade perca seu estado interno ou simplesmente pare de funcionar. Com falhas desse tipo a unidade nunca produz uma resposta incorreta.

Omission Fault. Falhas por omissão, implicam que a unidade não responda a algumas entradas e responda a outras.

Timing Fault. Falhas de performance, fazem com que a unidade responda adiantado ou atrasado.

Byzantine Fault. Falhas bizantinas, fazem com que a unidade se comporte de uma maneira totalmente arbitrária enquanto falha.

Essas falhas formam uma hierarquia, sendo as Falhas Crash as mais simples e as mais restritivas, e as Falhas Bizantinas as menos restritivas. Elas possuem uma relação de inclusão como mostrado na figura 1.1. Essa relação segue diretamente as definições de cada um dos tipos de falhas.

Diversos modelos têm sido propostos na literatura com o intuito de modelar um sistema distribuído para possibilitar a realização do diagnóstico dessas

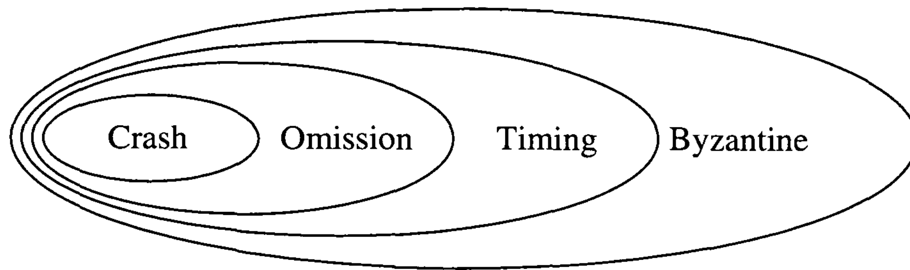

Figura 1.1: Classificação de Falhas.

falhas.

1.2 Modelos de Diagnóstico

Um algoritmo de diagnóstico em nível de sistema estipula testes para as unidades realizarem sobre outras unidades do sistema, sendo que, a partir dos resultados desses testes é descoberto o estado completo do sistema [5, 6, 16]. Nas propostas iniciais dos modelos para diagnóstico em nível de sistema, o diagnóstico é realizado por uma unidade especial do sistema, uma unidade ultra-confiável, chamada de supervisor central ou observador central [4, 16, 19, 20, 21, 23]. O supervisor, para realizar o diagnóstico, coleta os resultados dos testes de todos os processadores e analiza-os para descobrir quais unidades estão falhas e quais estão sem-falha.

Algoritmos que não necessitam da presença de um supervisor central no sistema, são ditos algoritmos distribuídos [15], neles as próprias unidades do

sistema realizam o diagnóstico ao invés de só o supervisor central.

Nos primeiros algoritmos propostos os testes eram fixos e definidos por uma unidade externa ao sistema, já os algoritmos nos quais as unidades do sistema decidem seus próximos testes baseadas nos resultados dos testes anteriores, são ditos algoritmos adaptativos [13].

O modelo PMC [21], proposto por Preparata, Metze e Chien, foi o primeiro modelo proposto, tendo introduzido o conceito de diagnóstico em nível de sistema. Esse é provavelmente o modelo de diagnóstico mais bem estudado. No modelo PMC, os testes são executados através de envios de estímulos e recebimento de respostas. A proposta inicial do modelo PMC define a existência de um supervisor central que especifica os testes a serem realizados, coleta os resultados desses e realiza o diagnóstico com base nessas respostas.

Uma abordagem alternativa ao modelo PMC, são os modelos baseados em comparações. Essa abordagem tem sua origem no modelo *Baseado em Comparações*, que foi proposto por Malek [20] e por Chwa e Hakimi [4]. Esse modelo baseia-se em comparações de resultados de tarefas para realizar o diagnóstico, ou seja, dois processadores executam uma mesma tarefa e as saídas geradas por eles, para essa tarefa, são comparadas e, com base nessas comparações, é realizado o diagnóstico do sistema. Alguns outros modelos utilizam-se da mesma técnica proposta pelo modelo Baseado em Comparações para executar o diagnóstico: o modelo MM [19], o modelo de

Comparações Generalizado [23] e o modelo *Broadcast Comparison* [3].

Algoritmos baseados nos dois modelos apresentados, os modelos baseados em comparações e o modelo PMC, fazem certas asserções com relação ao sistema. Uma asserção que esses modelos fazem é que os processadores sem-falha são confiáveis. Outra asserção é que o sistema é representável por um grafo completo, no qual os vértices são os processadores e as arestas são os *links* de comunicação. Como esse grafo é completo, deve existir uma aresta entre qualquer par de vértices, ou seja, todos os processadores devem estar interconectados. Outra asserção comum é que não se assume falhas de *links*, somente falhas de processadores.

Já os *modelos probabilísticos*[17], apresentam uma abordagem alternativa aos outros dois modelos. Esses modelos trabalham sobre probabilidades de respostas corretas. Os modelos probabilísticos não serão discutidos neste trabalho.

1.3 *Hierarchical Comparison-based Adaptive Distributed System-Level Diagnosis Algorithm*

Após o estudo dos diferentes modelos para diagnóstico em sistemas distribuídos, chegou-se à proposta deste trabalho: desenvolver um novo modelo para diagnóstico baseado em comparações e um algoritmo baseado nesse

modelo que realize o diagnóstico do sistema de forma hierárquica, adaptativa e distribuída.

O *Hierarchical Comparison-based Adaptive Distributed System-Level Diagnosis Algorithm (Hi-Comp)*, realiza seus testes baseado em comparações, para tanto, uma tarefa é enviada para duas unidades distintas do sistema. Essas unidades realizam a tarefa e devolvem os resultados para a testadora. Quando a testadora recebe os dois resultados, ela compara-os. Se os resultados forem iguais, significa que as duas unidades em questão estão sem-falha. Já se os resultados forem diferentes, alguma das duas unidades que realizou a tarefa está falha, mas o testador não consegue dizer qual delas.

O algoritmo *Hi-Comp* utiliza uma hierárquia de testes baseada na hierárquia utilizada pelos algoritmos *Hi-ADSD*, *Hi-ADSD with Detours* e *Hi-ADSD with Timestamps*, esses algoritmos são baseados no modelo PMC e serão detalhados no capítulo 2.

O algoritmo *Hi-Comp* é dito distribuído, pois as próprias unidades que fazem os testes realizam o diagnóstico, não sendo necessária a existência de um observador central no sistema. O algoritmo é dito adaptativo, porque as unidades baseiam-se nos resultados de testes já realizados para decidirem quais serão seus próximos testes. E o algoritmo é dito hierárquico pois os testes são realizados de forma hierárquica.

No algoritmo *Hi-Comp*, uma rodada de testes é definida como o tempo

necessário para que todas as unidades sem-falha do sistema obtenham informações de diagnóstico sobre todas as unidades do sistema. Graças à noção de rodada de testes e à forma como os testes são realizados, o novo algoritmo possui uma latência latência igual a log_2N rodadas de testes para um sistema de N nodos. Prova-se que o algoritmo é $(N-1)$-diagnosticável e que o número máximo de comparações executadas é de $O(N^3)$ no pior caso.

O restante desse trabalho está organizado da seguinte maneira. No capítulo 2 encontra-se um breve histórico sobre alguns modelos clássicos de diagnóstico, incluindo a revisão de alguns algoritmos de diagnóstico em nível de sistema. No capítulo 3 encontra-se a especificação do novo algoritmo; o capítulo 4 contém resultados de simulação do algoritmo e o capítulo 5, a conclusão.

Capítulo 2

Modelos de Diagnóstico em Nível de Sistema

A teoria de diagnóstico em nível de sistema tem recebido considerável atenção nas últimas quatro décadas. Em diagnóstico em nível de sistema busca-se identificar todas as unidades falhas de um sistema. Claramente isso pode não ser sempre possível, por exemplo, quando todas as unidades de um sistema estão falhas, nenhuma unidade pode ser utilizada para a realização do diagnóstico.

Para determinar quão diagnosticável um sistema é, e para realizar o diagnóstico, alguns modelos podem ser utilizados. O primeiro modelo proposto foi o *modelo PMC* [21] no qual os testes são executados através de envios de estímulos e recebimentos de respostas.

Os algoritmos distribuídos baseados no modelo PMC, que serão apresentados a seguir, possuem uma limitação, eles somente realizam o diagnóstico de falhas crash, tipo de falha onde os nodos simplesmente deixam de fun-

cionar, não provocando nenhuma ação incorreta no sistema. Alguns algoritmos baseados em outros modelos não possuem tal limitação e realizam o diagnóstico dos diversos tipos de falhas apresentados no capítulo 1. Esses algoritmos baseiam-se nos modelos baseados de comparações [3] e no modelo probabilístico [17]. Este trabalho considera apenas apenas o modelo PMC e os modelos baseados em comparações.

Os *modelos baseados em comparações* determinam que uma mesma tarefa seja enviada para duas unidades diferentes, os resultados dessa tarefa são recebidos e comparados. Se os resultados forem iguais, as duas unidades são consideradas sem-falha, e se eles forem diferentes, uma delas, pelo menos, está falha, mas não se consegue dizer qual delas. Poucos são os modelos baseados em comparações que conseguem lidar com as recuperações dos nodos falhos do sistema.

Neste capítulo será apresentada uma visão geral do modelo PMC e de alguns algoritmos baseados nele. Além disso, também são apresentados alguns modelos baseados em comparações.

2.1 Modelo PMC

O *modelo PMC* [21] foi o primeiro modelo de diagnóstico em nível de sistema proposto. Nesse modelo, um sistema S é decomposto em N unidades,

não necessariamente idênticas. Cada unidade é uma porção bem definida do sistema. As unidades não podem ser decompostas para propósito de diagnóstico, isto é, a unidade inteira deve ser considerada trabalhando corretamente, ou a unidade inteira deve ser considerada como falha. As unidades devem ser suficientemente poderosas para testarem outras unidades no sistema e determinar se elas estão falhas ou sem-falha.

Os algoritmos distribuídos baseados no modelo PMC, representam o sistema como um grafo, no qual as unidades são os vértices e as arestas são os *links* que interligam as unidades, sendo estes os caminhos por onde os testes são executados. Esses algoritmos não assumem falhas de *links*. Além disso, cada unidade do sistema deve estar interconectada a todas as demais, ou seja, deve existir um *link* de comunicação entre todos os pares de unidades do sistema. A figura 2.1 exemplifica um grafo representando um sistema de 4 unidades. Neste trabalho as unidades são também chamadas de *nodos*.

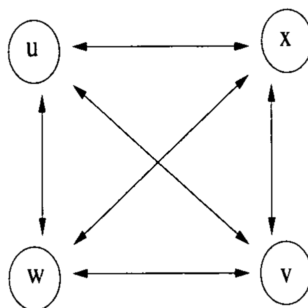

Figura 2.1: Grafo do Sistema.

Além do grafo do sistema também é definido o *grafo de testes* do sistema, que é um grafo direcionado cujos vértices são os nodos do sistema, e as arestas deste grafo são direcionadas de u para v, $u \rightarrow v$, quando o nodo u testa o nodo v. Para ilustrar essa situação considere o grafo de testes na figura 2.2, cada vértice representa um nodo e as arestas são direcionadas exatamente como os testes são executados, ou seja, o nodo u testa o nodo v, o nodo v testa o nodo w e o nodo w testa o nodo u.

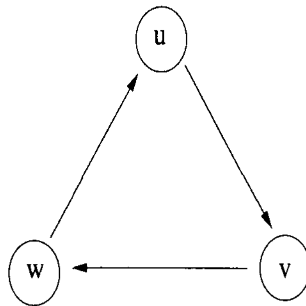

Figura 2.2: Grafo de Testes.

O diagnóstico no modelo PMC depende da capacidade dos nodos de testarem o estado de outros nodos. Assume-se que o estado de um nodo é *falho* ou *sem-falha* e que o estado de um nodo não muda durante o diagnóstico. Um teste envolve aplicações controladas de alguns estímulos e a observação das correspondentes respostas. Assume-se ainda, que um nodo sem-falha sempre reporta corretamente o estado de um nodo testado [12, 21], enquanto que nodos falhos podem retornar resultados incorretos dos testes

que ele realizou. A saída de um teste é simplesmente 1 (falho) ou 0 (sem-falha). Um teste *completo* sempre retorna o estado correto do nodo testado, isto é, o resultado de um teste completo sempre reflete a condição atual do nodo testado. A meta de diagnóstico é assinalar testes para os nodos executarem sobre outros nodos, e, quando os resultados desses testes forem corretamente analizados, obtenha-se o estado completo do sistema.

Uma saída $a_{i,j}$ é associada a cada teste que o nodo i executa sobre o nodo j, onde $a_{i,j}$ é a saída do respectivo teste. Se a unidade testadora i é sem-falha, $a_{i,j}$ é 0 se j é sem-falha, e 1 se j é falho. Se i é falho, então o resultado do teste não é confiável e $a_{i,j}$ pode assumir qualquer valor, independente do estado de j. O conjunto de saídas dos testes de um sistema S é chamado de *síndrome de* S [16].

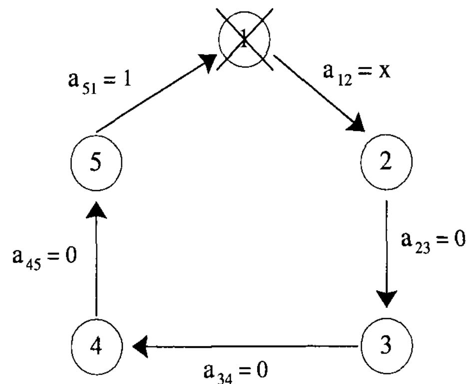

Figura 2.3: Síndrome de um sistema de 5 nodos.

Como exemplo considere o modelo do sistema da figura 2.3. O sistema

consiste de 5 nodos e os testes são atribuídos como mostrado na figura, isto é, o nodo 1 testa o nodo 2, o nodo 2 testa o nodo 3, e assim por diante. A síndrome desse sistema é um vetor de 5 bits:

$$(a_{1,2}; a_{2,3}; a_{3,4}; a_{4,5}; a_{5,1})$$

Considere, como mostrado na figura 2.3 que somente o nodo 1 está falho então a síndrome do sistema só pode ser:

$$(x; 0; 0; 0; 1).$$

Pois o nodo 2 identifica o nodo 3 como sem-falha, $a_{2,3} = 0$, o nodo 3 identifica o nodo 4 como sem-falha, $a_{3,4} = 0$, o nodo 4 identifica o nodo 5 como sem-falha, $a_{4,5} = 0$, já o nodo 5 identifica o nodo 1 como falho, $a_{5,1} = 1$, e, como o nodo 1 está falho, o valor de $a_{1,2}$ pode ser 0 ou 1, então é representado por um x.

Nesse modelo cada nodo conhece somente os resultados de seus testes. Para realizar o diagnóstico, as unidades falhas devem ser identificadas pelos resultados dos testes. No modelo PMC, a síndrome é analisada por um supervisor central, o qual é um nodo ultra-confiável. Este supervisor realiza o diagnóstico do estado do sistema, ou seja, identifica cada nodo como falho ou sem-falha. A realização do diagnóstico pelo supervisor nem sempre é trivial, pois os resultados dos testes de um nodo que foi declarado como falho por um outro nodo do sistema não podem ser utilizados para o diagnóstico.

Definição: Um sistema S é *t-diagnosticável* se, dada uma síndrome, todos os nodos falhos em S podem ser identificados, desde que o número de nodos falhos não exceda t [16].

Para um certo conjunto de testes, a diagnosticabilidade, isto é, a capacidade de completar o diagnóstico do sistema, pode ser limitada. Por exemplo, o sistema mostrado na figura 2.3 é 1-diagnosticável, mas não é 2-diagnosticável. Ele é 1-diagnosticável desde que um nodo falho pode sempre ser determinado pelo método a seguir. Se na *síndrome*, ou em uma de suas permutações cíclicas, uma string de 0's é seguida por um 1, então este 1 corretamente representa o nodo falho. Para compreender que o sistema não é 2-diagnosticável considere a situação onde ambos nodo 1 e nodo 2 estão falhos, e suponha que o nodo 2 retorne um 0 (ele pode retornar qualquer coisa). A síndrome desse sistema é indistinguível da síndrome do sistema mostrado na figura 2.3 com um nodo falho, e, portanto não pode ser corretamente diagnosticado.

Em sistemas t-diagnosticáveis, o problema de determinar t para um dado sistema, ou seja, determinar o número máximo de nodos que podem estar falhos, tal que o conjunto de unidades falhas possa ser **unicamente** identificado com base em alguma síndrome, é chamado de "Problema da Diagnosticabilidade".

Já o problema de determinar as unidades falhas de uma síndrome qual-

quer, dado que existam no máximo t unidades falhas, é chamado de "Problema de Diagnóstico". O Problema de Diagnóstico refere-se a realmente encontrar um algoritmo para diagnóstico de uma dada síndrome (desde que, é claro, o sistema possa ser diagnosticado). Uma caracterização completa de sistemas t-diagnosticáveis do modelo PMC pode ser encontrada em [12].

Além da abordagem original do modelo PMC com um supervisor central que coleta os resultados dos testes e decide o estado do sistema, algumas outras variações sobre como o diagnóstico é executado vêm sendo propostas: como *diagnóstico distribuído* [15], na qual cada unidade chega ao seu próprio diagnóstico do sistema testando seus vizinhos e obtendo desses os seus resultados de testes, e *diagnóstico adaptativo* [13], o qual propõe que o diagnóstico e os testes sejam intercalados e o próximo teste a ser realizado é uma função dos resultados obtidos até o momento, ou seja, os nodos baseiam-se nos resultados de seus testes anteriores para decidirem quais serão seus próximos testes.

2.1.1 *Adaptive-DSD*

O primeiro algoritmo adaptativo e distribuído baseado no modelo PMC é o algoritmo *Adaptive-DSD* (*Adaptive Distributed System-Level Diagnosis algorithm*) [2]. Aos N nodos do sistema que está rodando o *Adaptive-DSD* são

atribuídos identificadores sequênciais de 0 a $N - 1$. Os nodos sem-falha do sistema executam testes em outros nodos do sistema até encontrarem um outro nodo sem-falha, ou testarem todos os outros nodos como falhos. Os testes são executados sequencialmente, o nodo v testa o nodo $v + 1$, o nodo $v + 1$ testa o nodo $v + 2$ e assim por diante. Ao testar um nodo falho o testador passa a testar o próximo nodo, isto é, se o nodo v testar o nodo $v + 1$ como falho, o nodo v vai executar um teste sobre o nodo $v + 2$ e assim por diante até encontrar um nodo sem-falha.

Ao testar um nodo sem-falha, o testador vai obter informações de diagnóstico sobre o nodo testado e todas as informações de diagnóstico que o nodo testado já tenha obtido sobre os outros nodos do sistema. Na figura 2.4 temos um exemplo de um sistema de 8 nodos rodando o algoritmo *Adaptive-DSD*. Nessa figura o nodo 0 testa o nodo 1, como o nodo 1 está falho o nodo 0 vai testar o próximo nodo, ou seja, o nodo 2. Ao fazer isso, o nodo 0 testa um nodo sem-falha, então o nodo 0 vai obter informações de diagnóstico sobre todos os outros nodos do sistema, isto é, os nodos 3, 4, 5, 6, e 7, através deste nodo sem-falha.

Normalmente, os algoritmos de diagnóstico em nível de sistema são avaliados segundo três critérios [7, 8, 10]: a quantidade de informações transferidas a cada teste, a latência do algoritmo, ou seja, o tempo necessário para que todos os nodos sem-falha do sistema realizem o diagnóstico de um evento,

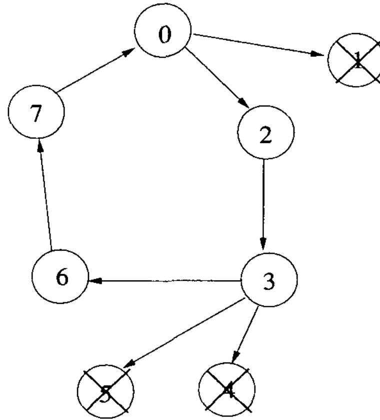

Figura 2.4: Sistema executando o algoritmo *Adaptive-DSD*.

e o número de testes executados a cada rodada de testes. Uma *rodada de testes* é definida como o período de tempo que todos os nodos sem-falha do sistema levam para testar um outro nodo sem-falha, ou testar todos os nodos do sistema como falhos [2, 10].

O algoritmo *Adaptive-DSD* apresenta algumas características: um nodo sem-falha do sistema será testado uma única vez a cada rodada de testes. Porém, um nodo sem-falha pode testar até $N-1$ nodos em uma única rodada de testes, isso ocorre quando o testador é o único nodo sem-falha do sistema e ele testa todos os nodos falhos, ou seja, $N-1$ nodos, a cada rodada de testes. Assim, ao considerarmos todos os nodos do sistema teremos, no máximo, N testes sendo executados a cada rodada de testes. A latência desse algoritmo é de N rodadas de testes, pois ao ocorrer um evento no nodo v a informação sobre esse evento deve passar por todos os $N-1$ nodos até chegar ao nodo

$v + 1$, pois a nova informação vai passar para um novo nodo a cada rodada de testes. Por exemplo, na figura 2.4, ao ocorrer um evento no nodo 7, esse evento vai ser diagnosticado pelo nodo 6 na próxima rodada de testes, pelo nodo 5 uma rodada de testes após o nodo 6 e assim por diante, até que o nodo 0 consiga diagnosticar esse evento. A quantidade de informações transferidas nesse algoritmo é sempre sobre $N - 1$ nodos, pois um nodo vai receber do nodo sem-falha testado informações sobre todos os outros nodos do sistema.

O algoritmo *Adaptive-DSD* consegue realizar o diagnóstico do sistema mesmo que exista somente um nodo sem-falha, o que faz com que o algoritmo seja classificado como $(N - 1)$-diagnosticável, assim como os algoritmos descritos a seguir.

2.1.2 *Hi-ADSD*

Em [10] Duarte e Nanya apresentam o algoritmo *Hi-ADSD* (*Hierarchical Adaptive Distributed System-Level Diagnosis algorithm*) que possui uma latência menor que a do algoritmo *Adaptive-DSD*. Nesse algoritmo os nodos são agrupados em conjuntos que são chamados de clusters. A quantidade de nodos de um cluster é sempre uma potência de dois, começando pelo menor com 1 nodo, ou seja, 2^0 nodos. O próximo cluster com 2 nodos, ou seja, 2^1 nodos, e os clusters seguintes vão aumentando de tamanho, até o maior, que

possui $N/2$ nodos, ou seja, 2^{logN-1} nodos. Todos os logaritmos deste trabalho possuem base 2. Na figura 2.5 temos a divisão de clusters do algoritmo *Hi-ADSD* para um sistema de oito nodos, por exemplo o nodo 0 é um cluster, o conjunto composto pelos nodos 2 e 3 é outro e o conjunto composto pelos nodos 4, 5, 6 e 7 é outro.

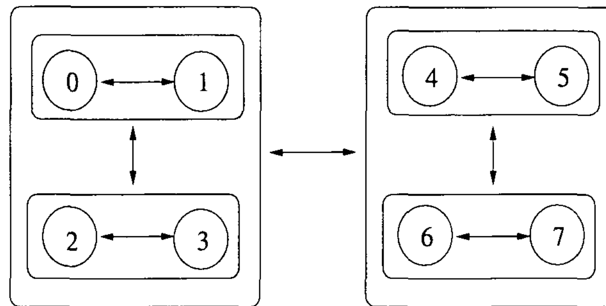

Figura 2.5: Divisão de clusters do algoritmo *Hi-ADSD*.

No algoritmo *Hi-ADSD* os nodos executam seus testes começando pelo menor cluster e prosseguindo para o segundo menor cluster, e assim por diante até testar o maior cluster. Após esse ser testado, o testador volta e começa a testar o menor cluster novamente, repetindo o ciclo. Ao testar um nodo sem-falha, o testador irá obter informações de diagnóstico sobre todos os nodos pertencentes ao cluster do nodo sem-falha testado. Porém ao testar um nodo falho, o nodo testador vai continuar testando nodos pertencentes ao cluster do nodo testado até testar um nodo sem-falha, ou testar todos os nodos desse como falhos. Se o testador testou todos os nodos de um cluster

como falhos, ele passa a testar o próximo cluster. Dessa forma um nodo vai executar testes no sistema até encontrar um nodo sem-falha, ou testar todos os outros $N - 1$ nodos do sistema como falhos. Por exemplo na figura 2.6 os nodos 2 e 3 estão falhos, o nodo 0 testa o nodo 2 como falho, então ele passa a testar o nodo 3, como esse também está falho, isto é, o cluster inteiro foi testado como falho, o nodo 0 passa a testar o primeiro nodo do próximo cluster, ou seja, testa o nodo 4, como esse está sem-falha, o nodo 0 irá obter, pelo nodo 4, informações de diagnóstico sobre todo este cluster, ou seja, sobre os nodos 4, 5, 6 e 7.

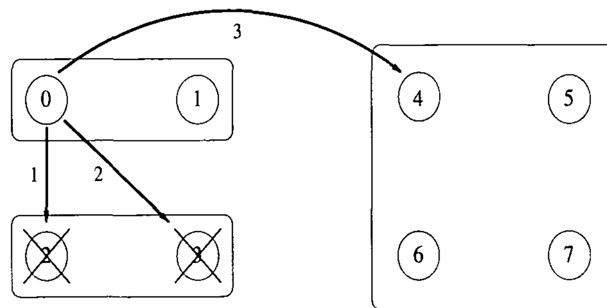

Figura 2.6: Sistema executando o algoritmo *Hi-ADSD*.

A latência do algoritmo *Hi-ADSD* é de, no máximo, $log^2 N$ rodadas de testes. Porém o número máximo de testes pode chegar à $N^2/4$ testes em uma única rodada de testes. Isso ocorre quando existem $N/2$ nodos sem-falha no sistema, e esses estão em um único cluster, além disso, é necessário que esses nodos estejam testando o outro cluster, com $N/2$ nodos falhos, na mesma

rodada de testes. Já a quantidade máxima de informações transferidas é sempre igual ao tamanho do cluster que está sendo testado. Um nodo pode chegar a transferir informações sobre $N/2$ nodos para cada teste a que for submetido.

2.1.3 *Hi-ADSD with Detours*

Em [7] Duarte, Brawerman e Albini apresentam o algoritmo *Hi-ADSD with Detours*, que apresenta uma melhora para o pior caso do número de testes do algoritmo *Hi-ADSD*, sem prejudicar a latência. Ele utiliza a mesma técnica de organizar os nodos em clusters que o *Hi-ADSD* utiliza, mas possui uma estratégia de testes diferente, que garante que o número máximo de testes é de $NlogN$ testes a cada $logN$ rodadas de testes. Nesse algoritmo, uma rodada de testes é definida como o período de tempo para que os nodos sem-falha realizem seus testes em um único cluster.

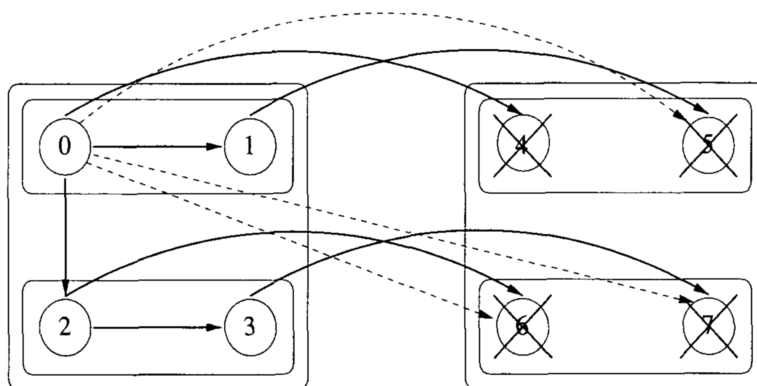

Figura 2.7: Sistema executando o algoritmo *Hi-ADSD with Detours*.

Para garantir a redução no número máximo de testes, os nodos que estão executando o algoritmo *Hi-ADSD with Detours* podem obter informações sobre os nodos de um cluster, contendo um nodo que foi testado como falho, por nodos de fora deste cluster, o que não é permitido no algoritmo *Hi-ADSD*.

Por exemplo, na figura 2.7 o nodo 0 vai obter informação sobre o nodo 5 pelo nodo 1 e sobre os nodos 6 e 7 pelo nodo 2, pois o nodo 4 está falho, com isso evita-se a existência de mais 3 testes sobre o cluster do nodo 4 que o nodo 0 faria se estivesse executando o algoritmo *Hi-ADSD*, os testes representados pelas linhas pontilhadas.

Essa mudança é possível graças à definição de *distância de diagnóstico*, que é a quantidade de arestas pelas quais as informações sobre o nodo u devem passar até atingir o nodo v, quando todos os nodos do sistema estão sem-falha. Por exemplo, na figura 2.8, a distância de diagnóstico entre o nodo 0 e o nodo 7 é 3, pois as informações do nodo 7, quando todos os nodos estão sem-falha, devem passar por três arestas até atingir o nodo 0.

A figura 2.8, ilustra o caminho normal que as informações do nodo 7 seguem até atingir o nodo 0, este caminho é o mesmo utilizado pelo algoritmo *Hi-ADSD*, ou seja, 0, 4, 6, 7, a linha contínua. Mas, se o nodo 4 está falho, pode-se pegar as informações sobre o nodo 7 através do caminho, 0, 2, 6, 7, linha pontilhada, que possui uma distância de diagnóstico igual a 3, ou

seja, a mesma que quando as informações são obtidas pelo caminho normal. Desta forma, o nodo 0 pode obter informações de diagnóstico sobre o nodo 7 por qualquer um dos dois caminhos com exatamente a mesma distância de diagnóstico, evitando assim, testes extras que o *Hi-ADSD* faz, sem atrasar o recebimento das informações. Esse outro caminho existente entre o nodo 0 e o nodo 7 é dito um *Detour*. *Detours* são caminhos alternativos entre dois nodos, e estes caminhos devem ter exatamente a mesma distância de diagnóstico que o caminho original.

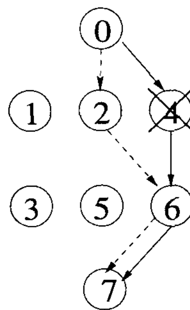

Figura 2.8: Caminho original e um *detour* entre o nodo 0 e o nodo 7.

2.1.4 *Hi-ADSD with Timestamps*

No algoritmo *Hi-ADSD with Timestamps* [8], busca-se o diagnóstico de *eventos*, ou seja, mudanças de estados nos nodos do sistema, de falho para sem-falha ou de sem-falha para falho, diferindo dos algoritmos anteriores, nos quais busca-se o diagnóstico de *estados* dos nodos, ou seja, os nodos sem-falha

determinam quais nodos do sistema estão falhos e quais estão sem-falha.

Além disso, no algoritmo *Hi-ADSD with Timestamps* é alterada a noção de cluster utilizada nos algoritmos anteriores. Neste algoritmo todos os clusters possuem $N/2$ nodos. A divisão dos nodos em clusters para um sistema de 8 nodos utilizando esse algoritmo é mostrada na figura 2.9, na qual cada cluster possui 4 nodos, pode-se notar que os nodos fazem parte de mais de um cluster, o que não acontecia com os nodos nos algoritmos anteriores, nos quais cada nodo só podia pertencer a um único cluster.

Nesse algoritmo, como nos anteriores, após um teste bem sucedido, o testador obtém do nodo testado infomações de diagnóstico sobre todo cluster do nodo testado, isto é, obtém informações de diagnóstico sobre $N/2$ nodos, sempre. Dessa forma um nodo v obtém informações de diagnóstico sobre o nodo u por diversos nodos e não somente por um nodo, adiantando assim o diagnóstico dos eventos.

Como o nodo v obtém informações de diagnóstico sobre o nodo u por diversos outros nodos é necessário saber qual das informações em questão é a mais nova, para evitar o diagnóstico de eventos antigos, essa é a motivação para o uso de informações datadas, ou seja, *Timestamps* [9, 22].

Resultados de simulação mostram que esse algoritmo reduz a latência média do diagnóstico de um evento. Por exemplo, para um sistema de 512 nodos, o *Hi-ADSD with Timestamps* reduz a latência de 40 rodadas de testes

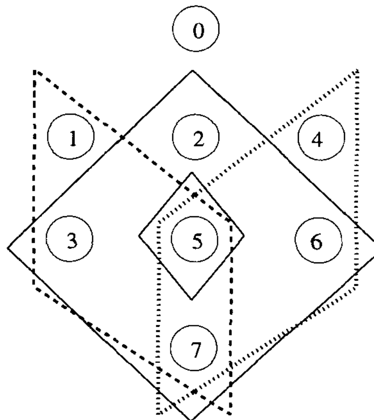

Figura 2.9: Divisão de clusters no *Hi-ADSD with Timestamps.*

para 12 rodadas de testes em média. Porém os piores casos da latência e do número de testes continuam os mesmos. Sendo que neste algoritmo, a quantidade de informações transferidas é maior, pois em todos os algoritmos hierárquicos apresentados, quando um nodo sem-falha testa outro nodo sem-falha, são transferidas informações de diagnóstico sobre todo o cluster do nodo testado, como no algoritmo *Hi-ADSD with Timestamps* os clusters possuem tamanho fixo a quantidade de informações transferida também é fixa, já nos algoritmos *Hi-ADSD* e *Hi-ADSD with Detours* a quantidade de informações trasferidas é proporcional ao tamanho do cluster.

2.2 Modelos de Diagnóstico Baseados em Comparações

O diagnóstico nos modelos baseados em comparações é realizado através da comparação de resultados de tarefas realizadas pelas unidades ou nodos do sistema. Para que isso aconteça, uma tarefa é enviada para, pelo menos, dois nodos do sistema, esses nodos executam essa tarefa e as saídas geradas são comparadas por outro nodo do sistema; se essas saídas forem iguais, o comparador considera que os dois nodos comparados estão sem-falha; mas se elas não forem iguais, o nodo que realizou a comparação das saídas não consegue dizer qual dos nodos que realizaram a tarefa está falho baseado somente nessa comparação.

Esses modelos possuem uma vantagem sobre os algoritmos distribuídos baseados no modelo PMC. Esses algoritmos somente conseguem realizar o diagnóstico de falhas do tipo crash, podendo deixar de diagnosticar falhas dos outros tipos existentes. Já nos modelos baseados em comparações é possível o diagnostico de falhas dos outros tipos.

2.2.1 Modelo Baseado em Comparações

O primeiro modelo de diagnóstico baseado em comparações publicado é o *modelo Comparison-Based* [20, 4] (modelo Baseado em Comparações), pro-

posto por Malek e por Hakimi e Chwa. Nesse modelo, os testes no sistema são executados através da duplicação de uma tarefa que deve ser executada em dois nodos do sistema. Após esses nodos executarem essa tarefa, suas saídas são enviadas para um nodo especial, ultra-confiável, o *observador central*. O observador central então compara as duas saídas e realiza o diagnóstico. Se as duas saídas forem iguais então os dois nodos que realizaram a tarefa são considerados sem-falha, porém, se as saídas não forem iguais, o observador central não sabe dizer qual dos nodos está falho, ele sabe que existe pelo menos um falho, mas não qual dos dois é o nodo falho.

Além disso, podem existir situações em que ambos os nodos que realizam uma mesma tarefa estão falhos, e a comparação realizada sobre essas duas saídas deve resultar em uma divergência. Para tanto o modelo estipula as seguintes asserções :

1. As saídas produzidas por dois nodos sem-falha para uma mesma tarefa sempre são idênticas;

2. Uma saída produzida por um nodo falho para uma tarefa tem que ser diferente de qualquer outra saída produzida para essa mesma tarefa, tanto por nodos sem-falha, como por outros nodos falhos.

Se os nodos do sistema respeitarem essas asserções, a única forma de duas saídas geradas por nodos diferentes para uma mesma tarefa serem idênticas

é quando os dois nodos estão sem-falha.

Mas esse modelo possui uma séria desvantagem: o observador central. Esse nodo é um nodo especial, ultra-confiável, que não pode sofrer nenhum evento, ou seja, ele não pode falhar em hipótese alguma. Além disso, é só o observador central que realiza o diagnóstico do sistema.

2.2.2 Modelo MM

O *modelo MM* [19] é um modelo de diagnóstico baseado em comparações que propõe que as comparações sejam feitas nos próprios nodos, eliminando com isso essa tarefa do observador central, ou seja, um nodo do sistema, não o observador central, vai coletar as saídas produzidas por dois outros nodos do sistema e compará-las. Após feita a comparação o resultado é enviado ao observador central para que o diagnóstico seja realizado.

O observador central analisa os resultados das comparações. Se o resultado de uma comparação constatou que as duas saídas eram idênticas e o comparador está sem-falha, o observador central sabe que os dois nodos que realizaram a tarefa estão sem-falha. Mas, se o resultado da comparação constatou que as duas saídas eram diferentes e o comparador está sem-falha, o observador central não consegue determinar qual dos dois nodos está falho. Além disso, como o comparador agora também é um nodo do sistema,

ele próprio pode estar falho. Nesse caso, nada é possível constatar sobre as comparações feitas por esse nodo.

No modelo MM, os sistemas são geralmente representados por multi-grafos como na figura 2.10b. *Multi-grafo* é um tipo de grafo, no qual pode existir diversas arestas ligando um mesmo par de nodos [11, 14, 23]. No multi-grafo 2.10b os vértices são os nodos do sistema e as arestas ligam pares de nodos que são comparados. No exemplo da figura 2.10a tem-se os links de comunicação do sistema e em 2.10b o multigrafo com as arestas interligando os pares de nodos que são comparados pelos outros nodos do sistema, os nodos comparadores estão indicados acima de cada aresta do grafo. Para existir uma aresta no grafo entre u e v, é necessário que exista um link de comunicação entre w e u e outro entre w e v, sendo que w será o comparador para a tarefa executada por u e por v.

Para distinguir entre os diferentes comparadores de um mesmo par de nodos u e v, o par de nodos (u, v) que foi comparado pelo nodo w é denotado por $(u, v)_w$, ou seja, w é o comparador para os nodos u e v. Para executar o diagnóstico, o comparador w vai comparar as saídas produzidas pelos dois nodos u e v quando submetidos a mesma tarefa; se as saídas forem idênticas o resultado da comparação será $r((u, v)_w) = 0$ caso contrário $r((u, v)_w) = 1$.

Considerando que o comparador pode estar falho, um resultado de comparação $r((u, v)_w) = 0$ implica que, se o comparador w está sem-falha, então

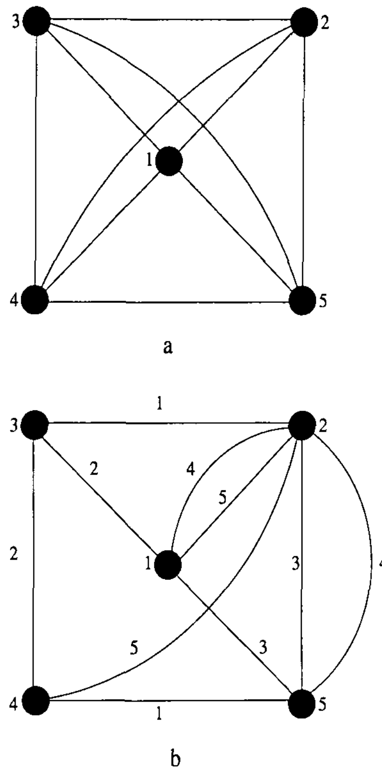

Figura 2.10: a) Topologia do sistema (arestas são links de comunicação).
b) Multigrafo do modelo MM (arestas são comparações).

os nodos u e v também estão sem-falha; por outro lado, se o resultado da comparação for $r((u,v)_w) = 1$ e o comparador w está sem-falha, estão sabe-se que o nodo u ou o nodo v ou até mesmo os dois, o nodo u e o nodo v, estão falhos; porém se o comparador w está falho, então nada se pode constatar de suas comparações.

Para realizar o diagnóstico o modelo MM utiliza-se das mesmas asserções que o modelo Baseado em Comparações e além disso ele impõe outras:

1. Todas as falhas são permanentes;

2. As comparações geradas por nodos falhos não são confiáveis;

3. Existe um limite no número de nodos falhos que podem existir no sistema.

Esse limite a que a asserção 3 se refere já foi explicado na seção 2.1, é a capacidade de completar o diagnóstico de um sistema, ou seja, um sistema é t-diagnosticável se existirem no máximo t unidades falhas no sistema de modo que duas síndromes diferentes não sejam confundidas.

Como exemplo, na figura 2.10 há uma determinada síndrome quando somente o nodo 2 está falho, e pode-se gerar a mesma síndrome quando ambos o nodo 2 e o nodo 3 estão falhos, portanto o sistema não é 2-diagnosticável, pois existem duas síndromes indistinguíveis. Mas o sistema é 1-diagnosticável, pois para somente um nodo falho todas as síndromes do sistema são corretamente identificadas.

2.2.3 Modelo de Comparações Generalizado

O *modelo de Comparações Generalizado* [23] apresenta uma generalização do modelo MM. Nesse modelo, o comparador, ou seja, o nodo que vai comparar as saídas produzidas para uma mesma tarefa, pode ser um dos dois nodos que realizam a tarefa. Por exemplo, é possível ter uma comparação $(u, v)_u$ ou $(u, v)_v$. Após executar a comparação o nodo comparador ainda envia

o resultado para o observador central que realiza o diagnóstico exatamente como no modelo MM.

Todo o restante do modelo de comparações generalizado é similar ao modelo MM.

2.2.4 Modelo *Broadcast Comparison*

O *modelo Broadcast Comparison* [3] é um modelo de diagnóstico baseado no modelo MM, porém nesse modelo o diagnóstico é distribuído e não mais centralizado como no modelo MM. Além disso, esse modelo assume que o sistema permite difusão confiável (*reliable broadcast*) [16] para comunicação entre os nodos do sistema.

Nesse modelo, pares de nodos são submetidos a uma mesma tarefa, ao finalizarem a execução desta tarefa, eles realizam difusão confiável de suas saídas para todos os nodos do sistema, ou seja, eles enviam as saídas das tarefas executadas para todos os nodos do sistema. Quando um nodo recebe os dois resultados, ele executa a comparação desses e constata se os nodos estão sem-falha ou se algum deles está falho, exatamente como no modelo MM. Porém como o diagnóstico é distribuído, os próprios nodos realizam o diagnóstico não sendo necessária a existência de um observador central no sistema.

Para garantir que o diagnóstico é correto são feitas certas asserções sobre o sistema e seu comportamento:

1. Uma comparação, executada por um nodo sem-falha, sobre resultados de tarefas realizadas por dois nodos sem-falha deve indicar uma igualdade;

2. Uma comparação, executada por um nodo sem-falha, sobre resultados de tarefas realizadas por um nodo falho e qualquer outro nodo, falho ou sem-falha, deve indicar uma desigualdade;

3. Qualquer menssagem de um nodo sem-falha que seja difundida de forma confiável no sistema é corretamente recebida por todos os outros nodos sem-falha dos sistema em um tempo definido;

4. O tempo para que uma tarefa produza uma saída é limitado;

5. Cada nodo tem uma identificação única;

6. Os nodos sem-falha podem identificar corretamente o emissor de uma difusão;

7. Mensagens transmitidas por nodos falhos e recebidas por nodos sem-falha não causam que a asserção 2 seja violada.

Porém, para a realização do diagnóstico, este modelo tem uma séria desvantagem, a realização da difusão confiável, que pode consumir uma quantidade considerável de recursos para ser realizada.

O modelo *Broadcast Comparison* garante que a capacidade de completar o diagnóstico de um sistema com N nodos é de $N - 1$ quando o grafo de comparações é completo, ou seja, existam arestas ligando qualquer par de nodos. Porém, quando o grafo não é completo a capacidade de completar o diagnóstico do sistema é menor, um sistema é t-diagnosticável se existirem pelo menos $t + 1$ nodos comparando cada nodo do sistema.

Capítulo 3

Um Algoritmo Hierárquico para Diagnóstico Distribuído baseado em Comparações

Neste capítulo é apresentado um novo modelo para diagnóstico distribuído baseado em comparações e um algoritmo hierárquico para diagnóstico baseado nesse modelo.

3.1 Modelo de Diagnóstico

No novo modelo, um sistema S é representado por um grafo $G = (V, E)$, onde V é o conjunto dos vértices e E é o conjunto das arestas. Os vértices do grafo representam os nodos do sistema e as arestas os *links* de comunicação. Neste modelo não se assume falhas de *links*. Os nodos do sistema S podem estar em um de dois estados *falho* ou *sem-falha*, e a mudança de estado de um nodo, de sem-falha para falho ou de falho para sem-falha, é dita um *evento*

no sistema.

O sistema S deve ser totalmente conectado, ou seja, deve existir um *link* de comunicação entre qualquer par de nodos do sistema. Assim sendo, o grafo G é um grafo completo, ou seja, $\forall i \in V$ e $\forall j \in V$, $\exists (i,j) \in E$.

Um nodo testa outros nodos do sistema para identificar seus estados. Um teste é realizado através do envio de uma tarefa para dois nodos distintos do sistema. Após a realização da tarefa, cada nodo envia a saída produzida para o nodo testador. O testador, ao receber os dois resultados produzidos para a mesma tarefa, compara-os. Se a comparação acusar que os resultados são idênticos, então o testador considera que os dois nodos que realizaram a tarefa estão sem-falha. Já se a comparação indicar que os resultados são diferentes, o testador sabe que pelo menos um dos dois nodos que realizaram a tarefa está falho, mas não pode concluir qual nodo está falho. Para garantir que os resultados das comparações são corretos, algumas asserções são feitas sobre o sistema. Estas asserções são descritas a seguir e têm sido utilizadas em diversos modelos de diagnóstico baseados em comparações [3, 19, 23]:

1. Uma comparação, executada por um nodo sem-falha, sobre resultados de tarefas realizadas por dois nodos sem-falha deve indicar a igualdade desses resultados.

2. Uma comparação, executada por um nodo sem-falha, sobre resultados

de tarefas realizadas por um nodo falho e qualquer outro nodo, falho ou sem-falha, deve indicar a desigualdade desses resultados.

3. O tempo para que um nodo sem-falha responda a uma tarefa é limitado.

Para garantir que a asserção 2 seja satisfeita, dois nodos falhos devem produzir saídas diferentes para uma mesma tarefa.

Quando um nodo sem-falha testa dois nodos sem-falha, ele obtém informação de diagnóstico de outros nodos do sistema, a partir dos nodos sem-falha testados.

3.2 O Algoritmo *Hi-Comp*

Nesta seção é apresentado um novo algoritmo para diagnóstico adaptativo distribuído e hierárquico baseado em comparações.

O novo algoritmo, *Hi-Comp (Hierarchical Comparison-based Adaptive Distributed System-Level Diagnosis algorithm)* é dito distribuído porque o diagnóstico é executado por todos os nodos do sistema, adaptativo porque os nodos baseiam-se em resultados de testes anteriores para decidirem os próximos testes, e é dito hierárquico porque os nodos adotam uma estratégia hierárquica de testes. O algoritmo é baseado no modelo apresentado na seção anterior.

O *Multi-grafo* [11, 14, 23] *de Testes do Sistema*, $T(S)$, é um multi-grafo direcionado definido sobre o grafo G, quando todos os nodos do sistema estão sem-falha. Os vértices de $T(S)$ são os nodos do sistema S. Já as arestas de $T(S)$ indicam que um nodo está enviando tarefas para outro nodo, ou seja, existe uma aresta direcionada do nodo i para o nodo j quando o nodo i envia uma tarefa para o nodo j.

Para diferenciar as arestas existentes entre um mesmo par de nodos, cada aresta possui um identificador. Esse identificador é constituído dos identificadores do par de nodos que esta aresta conecta, mais o identificador do outro nodo que realizou esta mesma tarefa e que terá sua saída comparada com a saída dessa tarefa. Desta forma, se o nodo i envia uma tarefa para ser realizada pelos nodos j e k, então existe no sistema uma aresta direcionada do nodo i para o nodo j identificada por $(i, j)_k$ e existe uma aresta direcionada do nodo i para o nodo k identificada por $(i, k)_j$. Assim sendo, se existe uma aresta $(i, j)_k$ direcionada do nodo i para o nodo j, então, deve, obrigatoriamente, existir uma aresta $(i, k)_j$ direcionada do nodo i para o nodo k.

Considere como exemplo a figura 3.1, como o nodo 1 envia tarefas para o nodo 2, para o nodo 3 e para o nodo 4 as arestas são: $(1, 2)_3$, $(1, 3)_2$, $(1, 2)_4$, $(1, 4)_2$, $(1, 3)_4$ e $(1, 4)_3$, todas direcionadas do nodo 1 para os outros nodos. A aresta $(1, 2)_3$ indica que o nodo 1 está enviando uma tarefa para o nodo 2

e a saída dessa será comparada com a saída do nodo 3, e, obrigatoriamente, deve existir a aresta $(1,3)_2$.

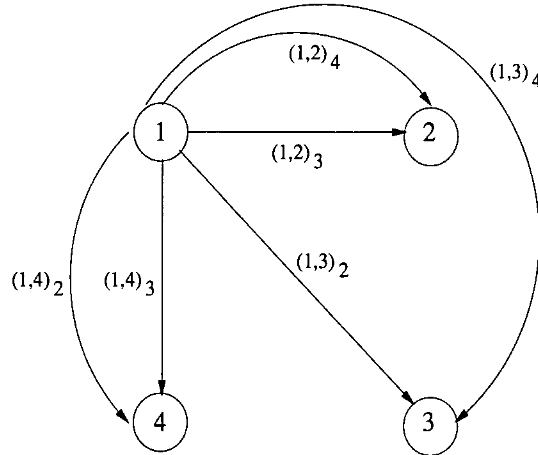

Figura 3.1: Multi-grafo T(S).

O *Grafo de Testes Livre de Falha*, TFF, é um grafo direcionado do sistema, definido sobre o multi-grafo $T(S)$. Para existir apenas uma aresta do nodo i para o nodo j em TFF é necessário que exista pelo menos uma aresta do nodo i para o nodo j em $T(S)$. Desta forma o grafo TFF é uma simplicação do multi-grafo $T(S)$ para um grafo simples, além disso, o grafo TFF é sempre um hipercubo quando a quantidade de nodos do sistema é uma potência de 2. A figura 3.2 representa o grafo TFF para o sistema representado pelo grafo T(S) da figura 3.1, e a figura 3.3 representa o grafo TFF para um sistema de 8 nodos.

A *Distância de Diagnóstico* entre o nodo i e o nodo j é definida como

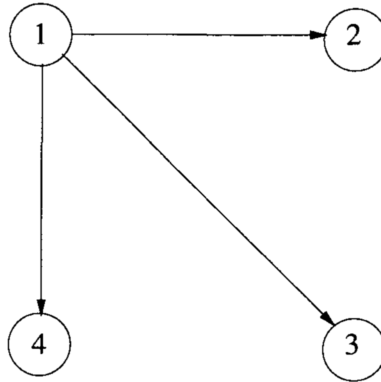

Figura 3.2: Grafo TFF.

a menor distância entre o nodo i e o nodo j em TFF, isto é, o caminho com o menor número de arestas entre o nodo i e o nodo j. Por exemplo, na figura 3.3 a distância de diagnóstico entre o nodo 0 e o nodo 5 é 2, pois o menor caminho entre esses nodos possui duas arestas. Pode existir mais de um caminho entre dois nodos com a mesma distância de diagnóstico.

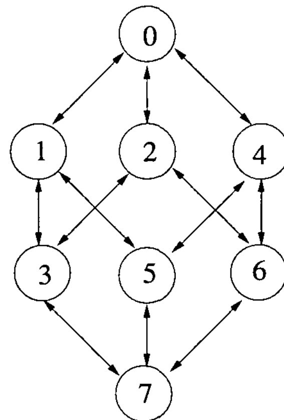

Figura 3.3: TFF para um sistema de 8 nodos.

O *Grafo de Testes Livre de Falhas do nodo i*, TFF_i, é um grafo direciona-

do, definido sobre o grafo TFF e representa a forma que as informações de diagnóstico fluem dentro do sistema. Em TFF_i existe uma aresta do nodo a para o nodo b se esta aresta existe em TFF e a distância de diagnóstico do nodo i para o nodo a for menor que a distância do nodo i para o nodo b. A figura 3.2 demonstra a TFF_0 para um sistema de 8 nodos. Nesta figura existe uma aresta do nodo 1 para o nodo 3 pois a distância de diagnóstico do nodo 0 para o nodo 1 é menor que a distância do nodo 0 para o nodo 3.

Uma *Rodada de Testes* é definida como o intervalo de tempo necessário para que todos os nodos sem-falha do sistema obtenham informações de diagnóstico sobre todos os nodos do sistema. Esta definição de rodada de testes é diferente daquela utilizada por outros algoritmos de diagnóstico distribuído apresentados no capítulo 3.

Sobre a definição de rodada de testes, é feita outra asserção ao sistema: Após o nodo i testar o nodo j em uma determinada rodada de testes, o nodo j não muda de estado nessa rodada de testes. Esta asserção se faz necessária para garantir a propagação das informações de diagnóstico pelo sistema, e ela é equivalente a asserção feita pelos algoritmos baseados no modelo PMC: Somente pode haver um novo evento no sistema, quando o evento anterior já tiver sido diagnosticado por todos os nodos sem-falha do sistema.

Os nodos que possuem ligação direta com o nodo i em TFF_i são ditos filhos de i. No exemplo da figura 3.6 os filhos do nodo 0 são os nodos 1, 2 e

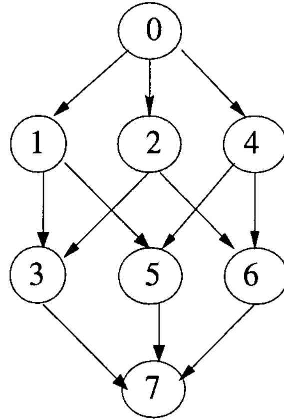

Figura 3.4: TFF_0 para um sistema de 8 nodos.

4.

A estratégia de testes agrupa os nodos em clusters, da mesma forma que no algoritmo *Hi-ADSD with Timestamps*, apresentado na seção 2.1.4. Uma função chamada $c_{i,s,p}$ define a lista de nodos dos quais o nodo i pode obter informação de diagnóstico através do nodo p com distância de diagnóstico, a partir de p, menor ou igual a $s-1$ em TFF_i. Em outras palavras, clusters são conjuntos de nodos dos quais o nodo i pode obter informações partindo de seus filhos, com distância de diagnóstico menor ou igual a $logN - 1$ para os seus filhos.

Nos clusters do novo algoritmo s é sempre igual a $logN$, o que implica que cada cluster tem sempre $N/2$ nodos. A figura 3.6 mostra a divisão de clusters para um sistema de 8 nodos em TFF_0. Os clusters são: (a)$c_{0,3,1}$: nodos 1, 3, 5 e 7, (b)$c_{0,3,2}$: nodos 2, 3, 6 e 7 e (c)$c_{0,3,4}$: nodos 4, 5, 6 e 7.

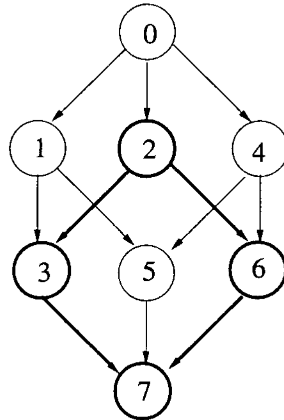

Figura 3.5: Um exemplo de $c_{i,s,p}$: $c_{0,3,2}$.

3.2.1 Descrição do Algoritmo

No novo algoritmo, em contraposição ao diagnóstico de estados, busca-se o diagnóstico de eventos. No algoritmo *Hi-Comp* os testes são feitos através do envio de uma tarefa para dois nodos distintos que a executam e devolvem os resultados para o nodo testador.

Inicialmente o nodo i envia uma tarefa para seus dois primeiros filhos, após receber os resultados desta tarefa, o nodo i envia outra tarefa para seus próximos dois filhos, e assim por diante até ter enviado tarefas para todos os seus filhos. Por exemplo, quando os filhos de 0 estão sem-falha, o nodo 0 envia a primeira tarefa para os nodos 1 e 2, seus dois primeiros filhos; depois ele envia outra tarefa para seus próximos dois filhos, os nodos 4 e 8, e assim por diante até ter enviado tarefas para todos os seus filhos.

Quando o nodo i diagnosticar que dois de seus filhos estão sem-falha, ele

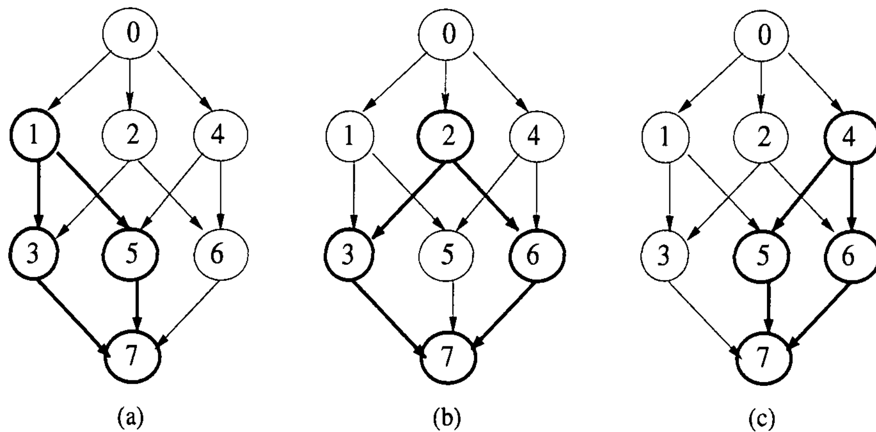

Figura 3.6: Clusters de um sistema de 8 nodos.

obtém informações de diagnóstico do cluster aos quais os nodos testados sem-falha pertencem. Por exemplo, na figura 3.6 quando o nodo 0 diagnosticar os nodos 1 e 2 como sem-falha, ele obtém pelo nodo 1 informações de diagnóstico de todo o cluster do nodo 1, ou seja, do nodo 3, do nodo 5 e do nodo 7, e pelo nodo 2 ele obtém informações de diagnóstico de todo o cluster do nodo 2, ou seja, do nodo 3, do nodo 6 e do nodo 7.

Neste algoritmo é possível que o nodo i obtenha informações sobre o nodo j através de dois ou mais nodos p e p', como ilustrado no exemplo acima onde o nodo 0 pode obter informação sobre o nodo 3 pelos nodos 1 e 2. Nesse caso é necessário garantir que o nodo i esteja sempre de posse da informação mais recente sobre os outros nodos do sistema. Para garantir isso, adotamos o uso de *timestamps* [8]. Essa estratégia é um mecanismo que permite que as informações sejam datadas. Os *timestamps* empregados são implementados

como contadores de trocas de estados que cada nodo possui para cada nodo do sistema.

Assim quando o nodo i obtém informação de diagnóstico do nodo j pelo nodo p, ele compara seu *timestamp* sobre o nodo j com o *timestamp* do nodo p sobre o nodo j, se o do nodo p for maior o nodo i atualiza suas informações pelas informações recebidas; caso contrário, as informações recebidas são descartadas.

Quando o nodo i executar uma comparação e esta indicar uma desigualdade, o nodo i classifica o estado dos dois nodos em questão como *indefinido*, pois ele não consegue saber qual dos nodos está falho ou sem-falha. Neste ponto, se o nodo i já identificou algum nodo como sem-falha, ele testa os dois nodos indefinidos em questão com o nodo sem-falha, um de cada vez. Se o resultado desse teste indicar uma igualdade, o testador classifica o nodo indefinido como sem-falha, caso contrário, como falho. Entretanto, se nenhum nodo foi classificado como sem-falha ainda, os nodos continuam com o estado indefinido, até que algum nodo sem-falha seja encontrado e um novo teste possa ser executado. Se existirem outros nodos indefinidos, os dois nodos que acabaram de ser classificados como indefinidos são testados com todos os já classificados anteriormente como indefinidos. Se dois desses nodos estiverem sem-falha, eles são encontrados; nesse caso, os demais estão todos falhos.

Se um nodo testar todos os seus filhos como indefinidos, ele deve passar

a testar os filhos dos seus filhos em TFF_i, e assim por diante até chegar ao último nodo, que deve ser testado com todos os nodos do sistema. Se o testador identificar algum nodo sem-falha, ele identifica, por este nodo, todos os nodos já classificados como indefinidos até o momento, seja obtendo informações de diagnóstico através do nodo sem-falha, seja testando esse nodo com os indefinidos.

Se, mesmo após utilizar o último nodo, nenhum nodo sem-falha foi encontrado. O testador assume que ele próprio está sem-falha e executa testes entre os nodos considerados indefinidos e ele mesmo. As comparações que derem indefinidas indicam que o nodo indefinido em questão está falho. Já as comparações que indicarem uma igualdade, indicam que o nodo indefinido em questão está sem-falha.

3.2.2 Especificação do Algoritmo

O novo algoritmo agrupa os nodos em três conjuntos: o conjunto dos nodos cujos estados estão indefinidos: U, o conjunto dos nodos falhos: F, e o conjunto dos nodos sem-falha: FF. Esses conjuntos são sempre disjuntos e a união deles resulta sempre em V, isto é, $U \cap F = \emptyset$, $U \cap FF = \emptyset$, $F \cap FF = \emptyset$ e $U \cup F \cup FF = V$. Cada nodo do sistema mantém estes três conjuntos, que podem variar de nodo para nodo. Ao final de cada rodada de testes, o

conjunto U estará totalmente vazio.

Quando o nodo i compara as saídas de uma tarefa realizada pelos nodos p e p', e essa comparação indicar uma igualdade, o nodo i identifica os dois nodos, p e p', como sem-falha. O nodo i coloca os dois nodos em FF retirando-os do conjunto ao qual pertenciam. Em outras palavras:

```
envia_tarefa(p,p');
SE (resposta(p) == resposta(p'))
ENTÃO
   U = U - {p};
   U = U - {p'};
   F = F - {p};
   F = F - {p'};
   FF = FF + {p} + {p'};
```

Quando o nodo i identifica um nodo sem-falha, ele obtém desse nodo informações de diagnóstico sobre todo o cluster desse nodo, ou seja, sobre os $N/2$ nodos daquele cluster. Além disso, como são utilizados *Timestamps* para datar as informações, o nodo i deve testar se as informações que está recebendo são mais novas que as que ele já possui. Em caso afirmativo, o nodo i deve atualizar suas informações locais pelas que está recebendo; caso contrário, deve simplesmente descartar essas informações. Em outras palavras:

```
envia_tarefa(p,p');
SE (resposta(p) == resposta(p'))
ENTÃO
   U = U - {p};
   U = U - {p'};
   F = F - {p};
   F = F - {p'};
   FF = FF + {p} + {p'};

   OBTER informações de diagnóstico de p;

   SE (Timestamps indica nova informação)
```

```
ENTÃO atualizar informações locais;

OBTER informações de diagnóstico de p';

SE (Timestamps indica nova informação)
ENTÃO atualizar informações locais;
```

Se a comparação que o nodo i realiza indicar uma desigualdade em relação à saída produzida pelo nodos p e p' o nodo i classifica os estados desses nodos como indefinidos. O nodo i coloca esses nodos no conjunto U e retira-os de qualquer outro conjunto a que eles pertençam. Em outras palavras:

```
envia_tarefa(p,p');
SE (resposta(p) != resposta(p'))
ENTÃO
   FF = FF - {p};
   FF = FF - {p'};
   F = F - {p};
   F = F - {p'};
   U = U + {p} + {p'};
```

Quando o testador classifica um nodo como indefinido, ele vai colocar esse nodo no conjunto U. Antes disso, antes do nodo p ser colocado em U, o testador, nodo i, deve testar o nodo p com todos os nodos $k \in U$. Se todos estes testes derem indefinidos, então, e só então, o nodo i coloca o nodo p em U. Em outras palavras:

```
envia_tarefa(p,p');
SE (resposta(p) != resposta(p'))
ENTÃO
   REPETIR para todo k pertencente a U
      envia_tarefa(p,k);
   ATÉ (k == ultimo elemento de U);
   SE (p é indefinido para todo k)
   ENTÃO
      FF = FF - {p};
      F = F - {p};
      U = U + {p};

   REPETIR para todo k pertencente a U
      envia_tarefa(p',k);
```

```
   ATÉ (k == ultimo elemento de U);
   SE (p' é indefinido para todo k)
   ENTÃO
      FF = FF - {p'};
      F = F - {p'};
      U = U + {p'};
```

Graças às comparações do nodo p com os nodos $k \in U$, quando um nodo k é identificado como sem-falha, todos os outros nodos de U estão, com certeza, falhos. Em outras palavras:

```
envia_tarefa(p,p');
SE (resposta(p) != resposta(p'))
ENTÃO
   REPETIR para todo k pertencente a U
      envia_tarefa(p,k);
   ATÉ (resposta(p) == resposta(k) OU (k == ultimo elemento de U);
   SE (resposta(p) == resposta(k))
   ENTÃO
      U = U - {p};
      F = F - {p};
      FF = FF + {p};
      U = U - {k};
      FF = FF + {k};
      F = F + U;
   SENÃO
      FF = FF - {p};
      F = F - {p};
      U = U + {p};

   REPETIR para todo k pertencente a U
      envia_tarefa(p',k);
   ATÉ (resposta(p') == resposta(k) OU (k == ultimo elemento de U);
   SE (resposta(p') == resposta(k))
   ENTÃO
      U = U - {p'};
      F = F - {p'};
      FF = FF + {p'};
      U = U - {k};
      FF = FF + {k};
      F = F + U;
   SENÃO
      FF = FF - {p'};
      F = F - {p'};
      U = U + {p'};
```

Quando o nodo i acaba de testar seus filhos, ele observa se os conjuntos U e FF não estão vazios. Essa situação ocorre se o nodo i encontrou nodos sem-falha e, depois de tê-los encontrado, ele testou alguns nodos como indefinidos.

Esses nodos que estão indefinidos precisam ser diagnosticados como falhos ou sem-falha. Para tanto, o nodo i escolhe um nodo sem-falha e envia tarefas para este nodo e para os nodos que estão indefinidos, ou seja, o nodo i vai enviar tarefas para um nodo $j \in FF$ e um nodo $k \in U$. Se a comparação dos resultados dessa tarefa indicar uma igualdade o nodo i coloca o nodo k em FF, caso contrário em F, pois o nodo j já foi identificado previamente como sem-falha. Em outras palavras:

```
envia_tarefa(j,k);
SE (resposta(j) == resposta(k))
ENTÃO
   U = U - {k};
   FF = FF + {k};
SENÃO
   U = U - {k};
   F = F + {k};
```

Quando um nodo $k \in U$ é identificado como sem-falha, o nodo i recebe dele os $N/2$ itens de informação de diagnóstico do cluster $c_{i,logN,k}$.

Se o nodo i acabou de testar seus filhos e o conjunto U está vazio e ainda existem nodos que o nodo i não conseguiu obter informações, o nodo i deve testar estes nodos com algum outro nodo já previamente identificado como sem-falha nesta rodada de testes.

Caso o nodo i tenha acabado de testar seus filhos e o conjunto FF estiver vazio, ou seja, todos os filhos de i estão falhos, o nodo i passa a testar os filhos de seus filhos, a próxima linha no grafo TFF_i, e assim por diante até obter alguma comparação que indique uma igualdade, ou até chegar ao último nodo de TFF_i.

Se o nodo i chegar ao último nodo, p, de TFF_i, ele deve enviar tarefas para esse nodo e para todos os nodos $k \in U$, um de cada vez, realizando os procedimentos adequados antes de colocar um nodo em U. Se nenhuma dessas comparações indicar uma igualdade, o nodo i coloca o nodo p em U, caso contrário, como acima, ele atinge o diagnóstico. Em outras palavras:

```
REPETIR para todo k pertencente a U
   envia_tarefa(p,k);
ATÉ (resposta(p) == resposta(k) OU (k == ultimo elemento de U);
SE (resposta(p) == resposta(k))
ENTÃO
   U = U - {p};
   F = F - {p};
   FF = FF + {p};
   U = U - {k};
   FF = FF + {k};
   F = F + U;
SENÃO
   FF = FF - {p};
   F = F - {p};
   U = U + {p};
```

Se, mesmo após utilizar o último nodo, o conjunto FF continuar vazio, o nodo i assume que ele próprio está sem-falha e se testa todos os nodos $k \in U$ consigo próprio. As comparações que derem indefinidas indicam que o nodo k em questão está falho. Se alguma comparação indicar uma igualdade este nodo k deve ser colocado em FF e todo o restante de U deve ser colocado em F. Em outras palavras:

```
REPETIR para todo k pertencente a U
   envia_tarefa(i,k);
ATÉ (resposta(i) == resposta(k) OU (k == ultimo elemento de U);
SE (resposta(i) == resposta(k))
ENTÃO
   U = U - {k};
   FF = FF + {k};
   F = F + U;
SENÃO
   U = U - {k};
   F = F + {k};
```

Assim, ao final de cada rodada de testes, todo nodo i sem-falha terá o conjunto U vazio e todos os outros nodos do sistema nos conjuntos F e FF, isto é, $U = \emptyset$ e $F \cup FF = V$.

Abaixo o algoritmo é apresentado em pseudo-código.

```
Algorithm running at node i:

TO_TEST = {ALL NODES};
U = EMPTY; F = EMPTY; FF = EMPTY;

REPEAT FOREVER

   REPEAT
      p = next_pair_to_test; p' = next_pair_to_test;
      result = send_task_and_compare(p,p');

      IF (result == 0)    /* p and p' are tested fault-free */
      THEN
         U = U - {p, p'}; F = F - {p, p'}; FF = FF + {p, p'}; TO_TEST = TO_TEST - {p, p'};
         GET N/2 itens of diagnostic information from p and p';
         FOR each peace of information
            COMPARE timestamps;
            UPDATE local diagnostic information if necessary;

      ELSE                /* test p and p' are tested undefined */
         IF (FF != EMPTY)
         THEN
            result = send_task_and_compare(p, node_of_FF);
            IF (result == 0)
            THEN
               F = F - {p}; U = U - {p}; FF = FF + {p};
               GET N/2 itens of diagnostic information from p;
               FOR each peace of information
                  COMPARE timestamps;
                  UPDATE local diagnostic information if necessary;
            ELSE
               U = U - {p}; FF = FF - {p}; F = F + {p};
            END_IF;

            result = send_task_and_compare(p', node_of_FF);
            IF (result == 0)
            THEN
               F = F - {p'}; U = U - {p'}; FF = FF + {p'};
               GET N/2 itens of diagnostic information from p';
               FOR each peace of information
                  COMPARE timestamps;
                  UPDATE local diagnostic information if necessary;
            ELSE
               U = U - {p'}; FF = FF - {p'}; F = F + {p'};
            END_IF;

         ELSE            /* FF == EMPTY */
            REPEAT
               k = select_new_node_from(U);
               result = send_task_and_compare(p,k);
               IF (result == 0)
               THEN
                  F = F - {p}; U = U - {p}; U = U - {k}; FF = FF + {k}; FF = FF + {p};
                  F = F + U + {p'}; U = EMPTY;
               END_IF;
            UNTIL (U == EMPTY) OR (k == last_node_from(U));

            IF (U != EMPTY)
            THEN
               REPEAT
                  k = select_new_node_from(U);
                  result = send_task_and_compare(p',k);
```

```
              IF (result == 0)
              THEN
                 F = F - {p'}; U = U - {p'}; U = U - {k}; FF = FF + {k}; FF = FF + {p'};
                 F = F + U + {p}; U = EMPTY;
              END_IF;
           UNTIL (U == EMPTY) OR (k == last_node_from(U));
           U = U + {p};

           IF (result == 1)
           THEN
              U = U + {p'}
           END_IF;
        END_IF;
     END_IF;
UNTIL (test == ok) or (node_to_test == last_node);

IF (TO_TEST != EMPTY)
THEN
   m = select_node_from(FF);
   REPEAT
      n = select_node_from(TO_TEST);
      result = send_task_and_compare(m,n);
      IF (result == 0)
      THEN
         F = F - {n}; U = U - {n}; TO_TEST = TO_TEST - {n}; FF = FF + {n};
      ELSE
         FF = FF - {n}; U = U - {n}; TO_TEST = TO_TEST - {n}; F = F + {n};
      END_IF
   UNTIL (TO_TEST == EMPTY);
END_IF

IF (|U| = N-2)      /* Last Node from TFFi */
THEN
   l = last_node_from_TFFi;
   REPEAT
      k = select_new_node_from(U);
      result = send_task_and_compare(l,k);
      IF (result == 0)
      THEN
         F = F - {l}; U = U - {l}; U = U - {k}; FF = FF + {k}; FF = FF + {l}; F = F + U; U = EMPTY;
      END_IF;
   UNTIL (U == EMPTY) OR (k == last_node_from(U));

   IF (U != EMPTY)
   THEN
      U = U + {l};
   END_IF;
END_IF;

IF (|U| = N-1)      /* Tester itself */
THEN
   REPEAT
      k = select_new_node_from(U);
      result = send_task_and_compare(i,k);
      IF (result == 0)
      THEN
         U = U - {k}; FF = FF + {k}; F = F + U; U = EMPTY;
      ELSE
         U = U - {k}; F = F + {k};
      END_IF;
   UNTIL (U == EMPTY);
END_IF;
```

3.3 Provas

Nesta seção são apresentadas as provas rigorosas da latência, do número de testes e da diagnosticabilidade (*diagnosability*) do novo algoritmo.

Teorema 1 *Todos os nodos sem-falha executando o algoritmo Hi-Comp necessitam de, no máximo, $logN$ rodadas de testes para atingir o diagnóstico completo do sistema.*

Prova:

Considere um novo evento que ocorre no nodo a. Pela definição de rodada de testes, todos os filhos do nodo a, os nodos que têm distância de diagnóstico igual a 1 com relação ao nodo a, realizam o diagnóstico desse evento na primeira rodada de testes posterior ao evento. Considerando o grafo TFF_a, ilustrado pela figura 3.7, na primeira rodada de testes posterior ao evento, os filhos do nodo a, realizam o diagnóstico do evento que ocorre em a.

Já na segunda rodada de testes, os nodos que têm distância de diagnóstico igual a 2 com relação ao nodo a realizam o diagnóstico, recebendo informações de diagnóstico dos nodos com distância de diagnóstico igual a 1, ou testando o nodo a diretamente, caso os nodos com distância 1 estejam falhos. Em TFF_a, ilustrado pela figura 3.7, os nodos que são filhos dos filhos de a realizam o diagnóstico do evento em questão, recebendo informações dos filhos de a ou testando o próprio nodo a.

Assuma que o nodo i sem-falha com distância de diagnóstico igual a d com relação ao nodo a realiza o diagnóstico de um evento ocorrido no nodo a em d rodadas de testes.

Considere agora um nodo j com distância de diagnóstico igual a $d + 1$ até o nodo a. Pela definição de distância de diagnóstico, todo nodo com distância de diagnóstico igual a $d + 1$ com relação ao nodo a é filho de algum nodo com distância de diagnóstico igual a d em relação ao nodo a. Então o nodo j é filho de algum nodo i. Pela definição de rodada de testes, um nodo obrigatoriamente testa todos os seus filhos em cada rodada de testes, então o j testa o nodo i em todas as rodadas de testes.

Como o nodo j testa o nodo i em todas as rodadas de testes, o nodo j pode demorar uma rodada de testes para receber novas informações através do nodo i. Então, se o nodo i demora d rodadas de testes para realizar o diagnóstico do evento ocorrido em a e o nodo j demora mais uma rodada de testes para realizar o diagnóstico através do nodo i, o nodo j demora $d + 1$ rodadas de testes para realizar o diagnóstico do evento ocorrido em a. Portanto, para um nodo j realizar o diagnóstico de um evento ocorrido em um nodo a, com distância de diagnóstico entre eles de $d + 1$, o nodo j demora $d + 1$ rodadas de testes.

Concluindo, se a distância de diagnóstico entre dois nodos for x então um desses nodos demora até x rodadas de testes para realizar o diagnóstico

de um evento ocorrido no outro nodo. Ou seja, um nodo pode demorar x rodadas de testes para realizar o diagnóstico de um evento em um nodo com distância de diagnóstico x até ele.

A latência máxima do algoritmo ocorre entre os nodos com maior distância de diagnóstico do sistema. Pela definição de um hipercubo [1, 24] e de distância de diagnóstico já apresentada, a maior distância de diagnóstico entre dois nodos no sistema é de $logN$. Portanto a latência máxima do algoritmo é de $logN$ rodadas de testes. □

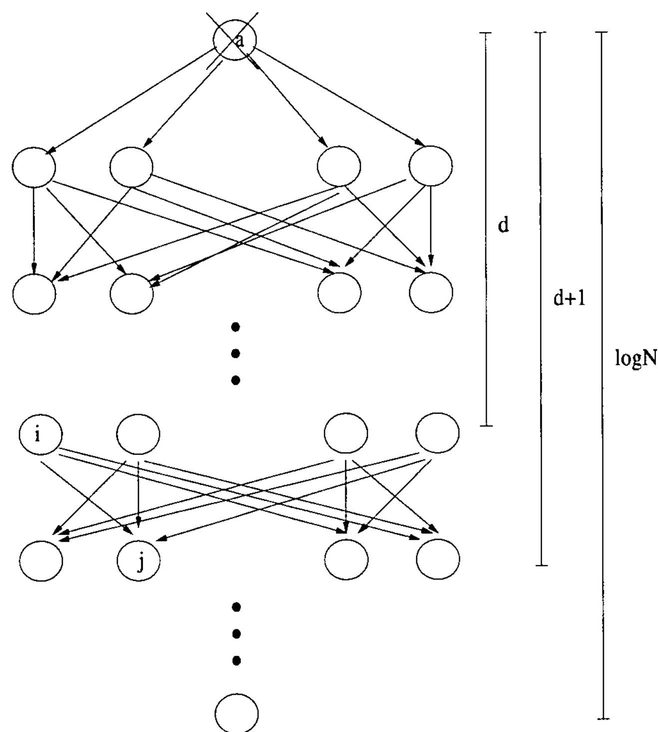

Figura 3.7: Ilustração de TFF_a.

Teorema 2 *O número máximo de testes realizados pelos nodos sem-falha em uma rodada de testes é $O(N^3)$.*

Prova:

Inicialmente, considere somente um nodo sem-falha no sistema e $N-1$ nodos falhos. Para realizar o diagnóstico, o nodo sem-falha envia testes para os nodos falhos combinando-os dois a dois, então o número de testes realizados é a combinação de $N-1$ nodos dois a dois: C^2_{N-1}

Porém somente com estes testes não é possível realizar o diagnóstico, então o nodo sem-falha assume que ele próprio está sem-falha, e envia tarefas para si próprio e para cada um dos nodos falhos, o que resulta na combinação de N nodos dois a dois: $C^2_N = \frac{N^2-N}{2}$

Agora considere dois nodos sem-falha. O pior caso para dois nodos sem-falha no sistema é atingido quando se tem o pior caso para cada um dos dois nodos. O número de testes executado neste caso é: $\frac{N^2-N}{2} + \frac{N^2-N}{2} = 2 * \frac{N^2-N}{2}$

Para três nodos sem-falha, o pior caso teórico é o pior caso para cada um dos nodos sem-falha: $3 * \frac{N^2-N}{2}$

Ao considerar-se N nodos sem-falha no sistema, o pior caso teórico para o número de testes é o pior caso para cada nodo: $N * (\frac{N^2-N}{2}) = \frac{N^3-N^2}{2}$, que é $O(N^3)$. □

Sabe-se que quanto maior a quantidade de nodos sem-falha no sistema, em geral, menos testes eles precisam realizar para atingir o diagnóstico, pois podem receber informações de diagnóstico de outros nodos sem-falha. Por exemplo, quando todos os N nodos estão sem-falha, cada nodo realiza $\frac{logN}{2}$ testes, um valor muito menor que $\frac{N^2-N}{2}$. Baseando neste fato e nos resultados de simulação obtidos, que são apresentados no próximo capítulo, supõe-se que o pior caso do número de testes deve ser menor que $O(N^3)$, sendo possivelmente $O(N^2)$, mas não há uma prova formal dessa suposição.

Teorema 3 *Um sistema executando o algoritmo Hi-Comp, é $(N-1)$-diagnosticável.*

Prova:

Primeiro considere um sistema com somente um nodo sem-falha e $N-1$ nodos falhos. Como já explicado anteriormente, o nodo sem-falha vai testar todos os nodos combinando-os dois a dois, até que teste todos os nodos consigo próprio e realize o diagnóstico completo do sistema, identificando os estados de todos os nodos.

Agora considere um sistema com mais de um nodo sem-falha. Um desses nodos sem-falha realiza testes até encontrar outros dois nodos sem-falha. Quando encontrar dois nodos sem-falha o testador obtém informações de diagnóstico dos dois nodos testados. Juntando as informações recebidas dos

nodos sem-falha com as informações coletadas pelos seus próprios testes, o nodo sem-falha realiza um diagnóstico completo do sistema.

Entretanto, se ocorrer a situação ilustrada pela figura 3.8, na qual o nodo a obtém informações de diagnóstico sobre o nodo c através do nodo b e, o nodo b obtém informações de diagnóstico sobre o nodo c através do nodo a, os nodos a e b não completariam o diagnóstico do sistema.

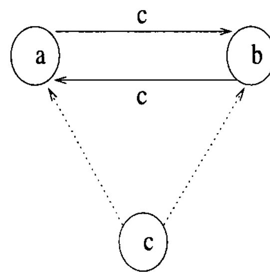

Figura 3.8: Caso impossível em um sistema executando o algoritmo *Hi-Comp.*

Porém essa situação jamais ocorre, pois para o nodo a receber informações do nodo c, através do nodo b, a distância de diagnóstico do nodo a para o nodo c deve ser maior que a distância do nodo b para o nodo c. Já para o nodo b receber informações sobre o nodo c através do nodo a, a distância do nodo b para o nodo c deve ser maior que a distância do nodo a para o nodo c. O que resulta em uma contradição, e nunca ocorre no sistema.

Desta forma, mesmo se houver apenas um único nodo sem-falha, esse nodo será capaz de completar corretamente o diagnóstico do sistema, assim sendo, o algoritmo é considerado $N-1$-diagnosticável. □

Capítulo 4

Resultados de Simulação

Neste capítulo são apresentados resultados experimentais obtidos através de simulações do algoritmo *Hi-Comp*. As simulações foram conduzidas utilizando a linguagem de simulação de eventos discretos SMPL [18]. Os nodos foram modelados como *facilities* do SMPL. Três tipos de eventos foram definidos: teste, falha e reparo.

Resultados de quatro experimentos são apresentados. O primeiro experimento ilustra a realização do diagnóstico de um evento pelo algoritmo *Hi-Comp* em um sistema de 16 nodos em duas situações: na primeira todos os nodos do sistema estão sem-falha quando ocorre o evento; na segunda situação, apenas dois nodos estão sem-falha e um deles sofre o evento; em ambos os casos é ilustrado como os nodos sem-falha realizam o diagnóstico dos eventos em questão. No segundo experimento, os nodos apresentam-se

em uma composição que leva à execução do algoritmo com o pior caso da latência, os resultados deste experimento vêm comprovar o teorema 1. Para o terceiro experimento foram examinadas todas as composições do sistema, o que permitiu a descoberta das situações que apresentam o maior número de testes por rodada de testes, partindo de um nodo sem-falha até N nodos sem-falha no sistema. No quarto experimento é demonstrada a diagnosticabilidade (*diagnosability*) do sistema.

4.1 Funcionamento do Algoritmo

Nesta seção são apresentados dois experimentos que detalham o funcionamento do algoritmo. Ambos foram realizados em um sistema de 16 nodos. No primeiro experimento todos os 16 nodos estão sem-falha e ocorre uma falha no nodo 15, o experimento ilustra como os nodos realizam o diagnóstico desse evento. No segundo algoritmo, somente os nodos 0 e 15 estão sem-falha, então ocorre um evento no nodo 15, o experimento ilustra o funcionamento do algoritmo para o diagnóstico desse evento em $log_2 16 = 4$ rodadas de testes.

4.1.1 $N-1$ Nodos Sem-Falha

Este experimento foi realizado em um sistema de 16 nodos, onde todos os nodos estão sem-falha, e o nodo 15 sofre um evento e torna-se falho, a situação do sistema é apresentada na figura 4.1.

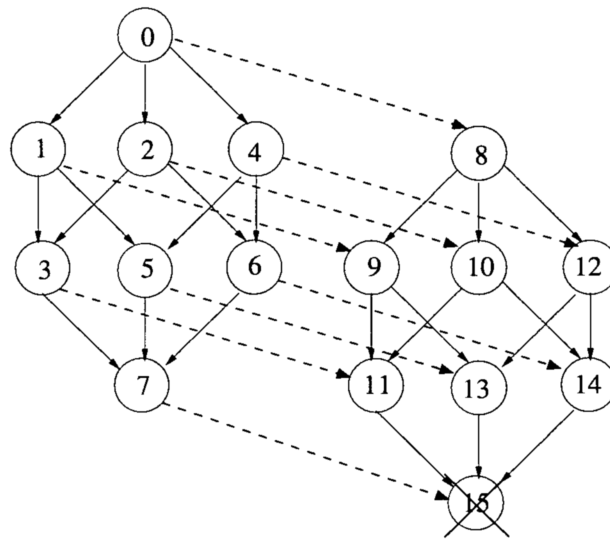

Figura 4.1: Sistema com 16 nodos com 1 evento.

O diagnóstico é ilustrado pela perspectiva do nodo 0 pois ele é o nodo mais distante do evento, e será o último a realizar o diagnóstico o evento. Na primeira rodada posterior ao evento, os nodos que possuem distância de diagnóstico igual a 1 para o nodo 15 conseguem diagnosticar o evento, ou seja, os nodos dos quais o nodo 15 é filho, isto é, os nodo 7, 11, 13 e 14. Nessa rodada de testes o nodo 0 vai testar seus filhos e obter informações de diagnóstico deles, porém as informações que ele recebe sobre o nodo 15 são informações antigas pois os filhos do nodo 0 ainda não diagnosticaram o

evento em questão.

Na segunda rodada após o evento, os nodos com distância de diagnóstico igual a 2 descobrem o evento quando estes testarem seus filhos, os nodos 7, 11, 13 e 14, pois estes nodos já possuem informações sobre o evento em questão e repassam essas informações quando são testados. Então, na segunda rodada, os nodos 3, 5, 6, 9, 10 e 12 realizam o diagnóstico do evento.

Na terceira rodada, os nodos com distância igual a 3 realizam o diagnóstico obtendo informações através dos nodos com distância igual a 2. Então, na terceira rodada, os nodos 1, 2, 4 e 8 recebem informações sobre o evento em questão.

Na quarta rodada, os nodos com distância de diagnóstico igual a 4 realizam o diagnóstico do evento, neste caso o nodo 0.

Portanto, em $log_2 16 = 4$ rodadas de testes todos os nodos sem-falha do sistema realizam o diagnóstico do evento.

4.1.2 1 Nodo Sem-Falha

Neste experimento existem $N - 2$ nodos falhos e somente os nodos 0 e 15 sem-falha, então ocorre um evento no nodo 15 que fica falho, a figura 4.2 ilustra a situação do sistema em questão.

O nodo 0 realiza todos os testes que um único nodo pode realizar. A

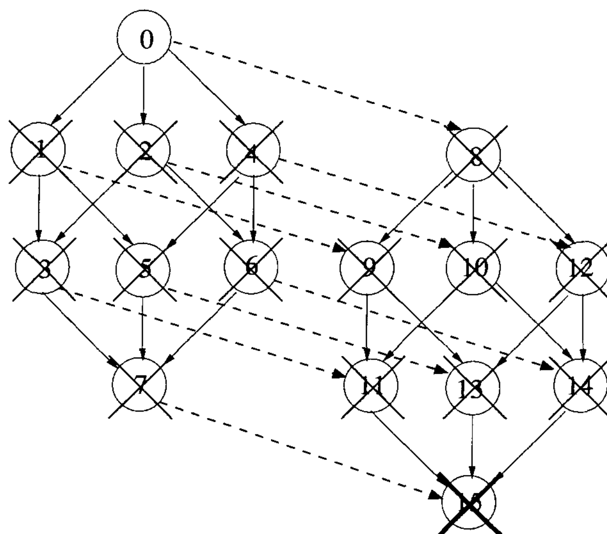

Figura 4.2: Sistema de 16 nodos com 1 evento e somente um nodo sem-falha.

saída do programa de simulação é apresentada abaixo:

```
node 15 becomes faulty at time 200.0
node 0 tests nodes 1 - 2 -> undefined at time 210.0.
node 0 tests nodes 4 - 8 -> undefined at time 210.0.
node 0 tests nodes 4 - 1 -> undefined at time 210.0.
node 0 tests nodes 4 - 2 -> undefined at time 210.0.
node 0 tests nodes 8 - 1 -> undefined at time 210.0.
node 0 tests nodes 8 - 2 -> undefined at time 210.0.
node 0 tests nodes 3 - 5 -> undefined at time 210.0.
node 0 tests nodes 3 - 1 -> undefined at time 210.0.
node 0 tests nodes 3 - 2 -> undefined at time 210.0.
node 0 tests nodes 3 - 4 -> undefined at time 210.0.
node 0 tests nodes 3 - 8 -> undefined at time 210.0.
node 0 tests nodes 5 - 1 -> undefined at time 210.0.
node 0 tests nodes 5 - 2 -> undefined at time 210.0.
node 0 tests nodes 5 - 4 -> undefined at time 210.0.
node 0 tests nodes 5 - 8 -> undefined at time 210.0.
node 0 tests nodes 6 - 9 -> undefined at time 210.0.
node 0 tests nodes 6 - 1 -> undefined at time 210.0.
node 0 tests nodes 6 - 2 -> undefined at time 210.0.
node 0 tests nodes 6 - 3 -> undefined at time 210.0.
node 0 tests nodes 6 - 4 -> undefined at time 210.0.
node 0 tests nodes 6 - 5 -> undefined at time 210.0.
node 0 tests nodes 6 - 8 -> undefined at time 210.0.
node 0 tests nodes 9 - 1 -> undefined at time 210.0.
node 0 tests nodes 9 - 2 -> undefined at time 210.0.
node 0 tests nodes 9 - 3 -> undefined at time 210.0.
node 0 tests nodes 9 - 4 -> undefined at time 210.0.
node 0 tests nodes 9 - 5 -> undefined at time 210.0.
node 0 tests nodes 9 - 8 -> undefined at time 210.0.
```

```
node 0 tests nodes 10 - 12 -> undefined at time 210.0.
node 0 tests nodes 10 - 1 -> undefined at time 210.0.
node 0 tests nodes 10 - 2 -> undefined at time 210.0.
node 0 tests nodes 10 - 3 -> undefined at time 210.0.
node 0 tests nodes 10 - 4 -> undefined at time 210.0.
node 0 tests nodes 10 - 5 -> undefined at time 210.0.
node 0 tests nodes 10 - 6 -> undefined at time 210.0.
node 0 tests nodes 10 - 8 -> undefined at time 210.0.
node 0 tests nodes 10 - 9 -> undefined at time 210.0.
node 0 tests nodes 12 - 1 -> undefined at time 210.0.
node 0 tests nodes 12 - 2 -> undefined at time 210.0.
node 0 tests nodes 12 - 3 -> undefined at time 210.0.
node 0 tests nodes 12 - 4 -> undefined at time 210.0.
node 0 tests nodes 12 - 5 -> undefined at time 210.0.
node 0 tests nodes 12 - 6 -> undefined at time 210.0.
node 0 tests nodes 12 - 8 -> undefined at time 210.0.
node 0 tests nodes 12 - 9 -> undefined at time 210.0.
node 0 tests nodes 7 - 11 -> undefined at time 210.0.
node 0 tests nodes 7 - 1 -> undefined at time 210.0.
node 0 tests nodes 7 - 2 -> undefined at time 210.0.
node 0 tests nodes 7 - 3 -> undefined at time 210.0.
node 0 tests nodes 7 - 4 -> undefined at time 210.0.
node 0 tests nodes 7 - 5 -> undefined at time 210.0.
node 0 tests nodes 7 - 6 -> undefined at time 210.0.
node 0 tests nodes 7 - 8 -> undefined at time 210.0.
node 0 tests nodes 7 - 9 -> undefined at time 210.0.
node 0 tests nodes 7 - 10 -> undefined at time 210.0.
node 0 tests nodes 7 - 12 -> undefined at time 210.0.
node 0 tests nodes 11 - 1 -> undefined at time 210.0.
node 0 tests nodes 11 - 2 -> undefined at time 210.0.
node 0 tests nodes 11 - 3 -> undefined at time 210.0.
node 0 tests nodes 11 - 4 -> undefined at time 210.0.
node 0 tests nodes 11 - 5 -> undefined at time 210.0.
node 0 tests nodes 11 - 6 -> undefined at time 210.0.
node 0 tests nodes 11 - 8 -> undefined at time 210.0.
node 0 tests nodes 11 - 9 -> undefined at time 210.0.
node 0 tests nodes 11 - 10 -> undefined at time 210.0.
node 0 tests nodes 11 - 12 -> undefined at time 210.0.
node 0 tests nodes 13 - 14 -> undefined at time 210.0.
node 0 tests nodes 13 - 1 -> undefined at time 210.0.
node 0 tests nodes 13 - 2 -> undefined at time 210.0.
node 0 tests nodes 13 - 3 -> undefined at time 210.0.
node 0 tests nodes 13 - 4 -> undefined at time 210.0.
node 0 tests nodes 13 - 5 -> undefined at time 210.0.
node 0 tests nodes 13 - 6 -> undefined at time 210.0.
node 0 tests nodes 13 - 7 -> undefined at time 210.0.
node 0 tests nodes 13 - 8 -> undefined at time 210.0.
node 0 tests nodes 13 - 9 -> undefined at time 210.0.
node 0 tests nodes 13 - 10 -> undefined at time 210.0.
node 0 tests nodes 13 - 11 -> undefined at time 210.0.
node 0 tests nodes 13 - 12 -> undefined at time 210.0.
node 0 tests nodes 14 - 1 -> undefined at time 210.0.
node 0 tests nodes 14 - 2 -> undefined at time 210.0.
node 0 tests nodes 14 - 3 -> undefined at time 210.0.
node 0 tests nodes 14 - 4 -> undefined at time 210.0.
node 0 tests nodes 14 - 5 -> undefined at time 210.0.
node 0 tests nodes 14 - 6 -> undefined at time 210.0.
node 0 tests nodes 14 - 7 -> undefined at time 210.0.
node 0 tests nodes 14 - 8 -> undefined at time 210.0.
node 0 tests nodes 14 - 9 -> undefined at time 210.0.
node 0 tests nodes 14 - 10 -> undefined at time 210.0.
node 0 tests nodes 14 - 11 -> undefined at time 210.0.
```

```
node 0 tests nodes 14 - 12 -> undefined at time 210.0.

Node 0 tests nodes 15 - 1 -> undefined at time 210.0.
Node 0 tests nodes 15 - 2 -> undefined at time 210.0.
Node 0 tests nodes 15 - 3 -> undefined at time 210.0.
Node 0 tests nodes 15 - 4 -> undefined at time 210.0.
Node 0 tests nodes 15 - 5 -> undefined at time 210.0.
Node 0 tests nodes 15 - 6 -> undefined at time 210.0.
Node 0 tests nodes 15 - 7 -> undefined at time 210.0.
Node 0 tests nodes 15 - 8 -> undefined at time 210.0.
Node 0 tests nodes 15 - 9 -> undefined at time 210.0.
Node 0 tests nodes 15 - 10 -> undefined at time 210.0.
Node 0 tests nodes 15 - 11 -> undefined at time 210.0.
Node 0 tests nodes 15 - 12 -> undefined at time 210.0.
Node 0 tests nodes 15 - 13 -> undefined at time 210.0.
Node 0 tests nodes 15 - 14 -> undefined at time 210.0.

Node 0 uses itself
Node 0 identifies through node 0 -> node 1 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 2 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 3 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 4 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 5 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 6 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 7 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 8 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 9 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 10 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 11 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 12 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 13 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 14 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 15 is Faulty at time 210.0.

Final Result:
Node 0 => TESTER
Node 1 Faulty at time 210.0.
Node 2 Faulty at time 210.0.
Node 3 Faulty at time 210.0.
Node 4 Faulty at time 210.0.
Node 5 Faulty at time 210.0.
Node 6 Faulty at time 210.0.
Node 7 Faulty at time 210.0.
Node 8 Faulty at time 210.0.
Node 9 Faulty at time 210.0.
Node 10 Faulty at time 210.0.
Node 11 Faulty at time 210.0.
Node 12 Faulty at time 210.0.
Node 13 Faulty at time 210.0.
Node 14 Faulty at time 210.0.
Node 15 Faulty at time 210.0.
```

Como ilustrado acima, o nodo 0 vai testando todos os nodos dois a dois, como ele não consegue identificar nenhum par de nodos sem-falha, então ele testa todos os nodos consigo próprio e, assumindo que ele está sem-falha,

realiza o diagnóstico de todos os nodos do sistema em uma única rodada de testes, sendo necessária para isso, a realização de $\frac{N^2-N}{2} = 120$ testes em uma única rodada de testes.

4.2 Latência do Algoritmo

Para demonstrar a latência do algoritmo *Hi-Comp* são utilizados dois experimentos: o primeiro leva em consideração o diagnóstico de um evento, nesse experimento todos os nodos do sistema estão sem-falha, então ocorre um evento em um nodo. O segundo experimento leva em consideração o diagnóstico de $N-1$ eventos, para esse experimento todos os N nodos do sistema estão sem-falha, então ocorrem $N-1$ eventos simultâneos deixando somente um nodo sem-falha no sistema inteiro e esse nodo deve realizar o diagnóstico de todos esses eventos.

4.2.1 Diagnóstico de um Evento

O propósito deste experimento é ilustrar a quantidade de rodadas de testes necessárias para que um evento seja propagado em um sistema de 16 nodos, com todos os outros $N-1$ nodos sem-falha.

Pela definição do algoritmo, um nodo deve obter informações de diagnóstico sobre todos os nodos do sistema a cada rodada de testes, o que, no

sistema em questão, é equivalente a dizer que, um nodo vai ser testado por todos os nodos dos quais ele é filho a cada rodada de testes. Sendo assim, uma rodada depois de ocorrer um evento no nodo k, todos os nodos dos quais k é filho realizam o diagnóstico desse evento. Na rodada seguinte essa informação é repassada através dos nodos que já realizaram o diagnóstico desse evento e assim por diante, até que todos os nodos sem-falha tenham realizado o diagnóstico desse evento .

Rodada de Testes	*Completam o Diagnóstico*
1	4
2	6
3	4
4	1

Tabela 4.1: Quantidade de nodos que realizam o diagnóstico de um evento por rodada de testes, para um sistema de 16 nodos.

Neste experimeto leva-se em conta um sistema de 16 nodos representado pela figura 4.1, o evento ocorre no nodo 15 e as informações sobre esse evento devem ser propagadas até atingir o nodo 0. A tabela 4.1 ilustra a quantidade de nodos que descobrem o evento a cada rodada de testes. Na primeira rodada posterior ao evento, 4 nodos realizam o diagnóstico, na segunda 6 nodos, na terceira rodada outros 4 nodos realizam o diagnóstico e na quarta rodada somente um nodo realiza o diagnóstico.

O gráfico 4.3 ilustra a quantidade de rodadas de testes necessárias, para que os nodos sem-falha do sistema, realizem o diagnóstico do evento ocorrido

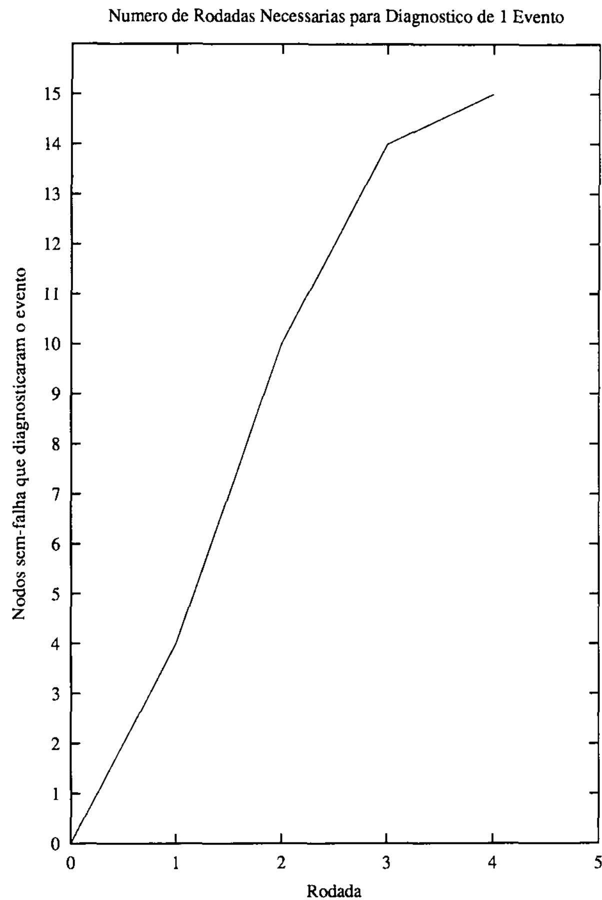

Figura 4.3: Rodadas de testes necesárias para todos os nodos sem-falha realizem o diagnóstico de 1 evento.

no nodo 15.

4.2.2 Um Nodo Realiza o Diagnóstico de $N-1$ Eventos

Neste experimento somente o nodo 0 está sem-falha e todos os outros nodos estão falhos, o sistema de 16 nodos é ilustrado pela figura 4.4. O nodo 0 já realizou o diagnóstico, e conhece o estado atual de todos os nodos do sistema, então acontece, ao mesmo tempo, um evento em cada nodo falho fazendo com

que todos os nodos do sistema fiquem sem-falha.

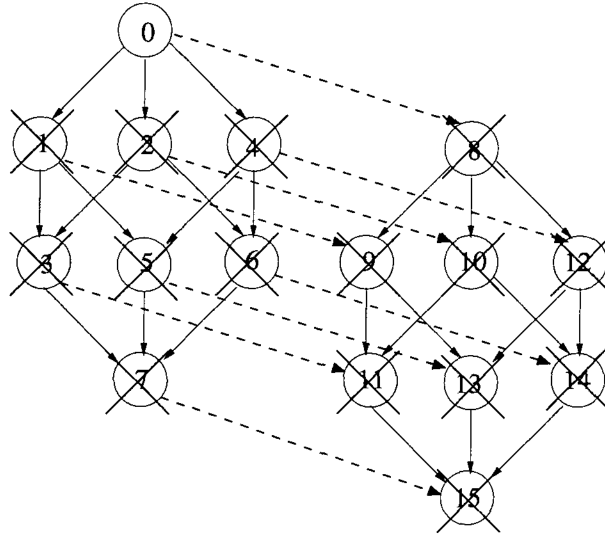

Figura 4.4: Sistema com 16 nodos com $N-1$ eventos.

Na primeira rodada de testes após os eventos, o nodo 0 realiza o diagnóstico dos eventos ocorridos em seus filhos, exatamente como todos os outros nodos do sistema, pois agora todos estão sem-falha e realizando o diagnóstico. Na segunda rodada o nodo 0 realiza o diagnóstico dos filhos de seus filhos através destes, e assim por diante até realizar o diagnóstico do sistema inteiro.

A tabela 4.2 demonstra a quantidade de eventos que o nodo 0 consegue diagnosticar por rodada de testes.

Portanto, em $log_2 16 = 4$ rodadas de testes após os eventos, o nodo 0 consegue realizar o diagnóstico de todos os eventos ocorridos.

Rodada de Testes	*Número de Nodos*
1	4
2	6
3	4
4	1

Tabela 4.2: Quantidade de eventos que o nodo 0 consegue diagnosticar por rodada de testes, para um sistema de 16 nodos.

4.3 Quantidade Máxima de Testes Necessários

O propósito deste experimento é mostrar a quantidade máxima de testes realizados pelos nodos sem-falha em uma rodada de testes. Para a realização desse experimento foram examinadas todas as composições de nodos possíveis para um sistema de 16 nodos, e escolhidas as que resultam nas maiores quantidades de testes para todas as quantidades de nodos sem-falha. Pode haver mais de uma composição, com a mesma quantidade de nodos sem-falha que levem a este pior caso, porém neste experimento não importa a composição e sim a quantidade de testes realizados.

A quantidade de nodos sem-falha no sistema começa em 0 e é incrementada até que os N nodos do sistema estejam sem-falha. Então foram selecionadas as situações onde os nodos sem-falha executam a maior quantidade de testes possível.

No gráfico 4.5, a linha contínua ilustra o número de testes realizados no sistema para as diferentes quantidades de nodos sem-falha; já a linha ponti-

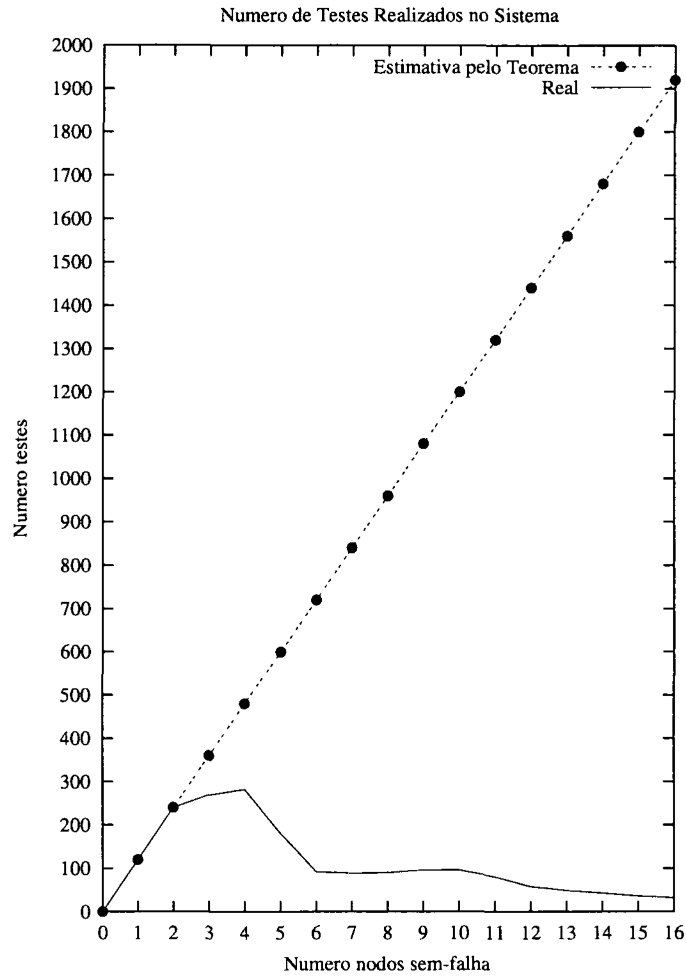

Figura 4.5: Número de Testes excutado no Sistema.

lhada ilustra qual deveria ser o pior caso do número de testes de acordo com o teorema 2. Como é ilustrado no gráfico, a quantidade máxima de testes real é muito menor que a quantidade teórica apresentada no teorema 2.

Como o gráfico 4.5 ilustra, a maior quantidade de testes no sistema simulado ocorre quando existem 4 nodos sem-falha. Esse caso ocorre, porque os nodos sem-falha estão arranjados como na figura 4.6. No algoritmo *Hi-*

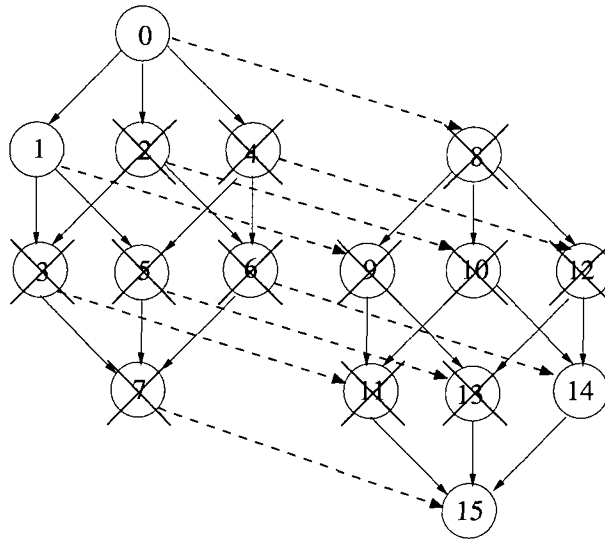

Figura 4.6: Pior caso do número de testes para 16 nodos.

Comp, um nodo realiza testes no sistema até descobrir dois nodos sem-falha. Como ilustrado na figura 4.6, os nodos foram arranjados de forma que devam realizar o maior número de testes para encontrarem dois nodos sem-falha. Por exemplo, no caso do nodo 0, ele precisa testar todos os nodos entre ele e o nodo 14, enviando testes para cada par de nodos nesse intervalo, até testar o par formado pelos nodos 1 e 14. Isso se repete para cada um dos quatro nodos, maximizando o número de testes executados no sistema.

Essa simulação vem comprovar as suspeitas de que o número máximo de testes realizados no sistema não é $O(N^3)$. Pois, como é demostrado no gráfico 4.5, a medida que a quantidade de nodos sem-falha aumenta, o número de testes diminui.

4.4 Diagnosticabilidade (*Diagnosability*)

O experimento "1 Nodo Sem-Falha" apresentado na seção 4.1.2 comprova que um sistema executando o algoritmo *Hi-Comp* é $N-1$-diagnosticável pois, como já apresentado, quando existe somente um nodo sem-falha no sistema, esse nodo consegue realizar o diagnóstico de todos os $N-1$ nodos falhos.

No experimento da seção 4.1.2 é apresentada a saída do programa de simulação para uma falha que ocorre no nodo 15, agora consideremos outro experimento: todos os nodos dos sistema estão sem-falha, então ocorrem $N-1$ eventos simultâneos, deixando somente o nodo 0 sem-falha. Com isso existem agora $N-1$ eventos que o nodo 0 deve diagnosticar, a saída do programa de simulação para essa situação é apresentada abaixo, e como pode-se notar ela é identica a saída já apresentada na seção 4.1.2.

```
node 1 becomes faulty at time 200.0
node 2 becomes faulty at time 200.0
node 3 becomes faulty at time 200.0
node 4 becomes faulty at time 200.0
node 5 becomes faulty at time 200.0
node 6 becomes faulty at time 200.0
node 7 becomes faulty at time 200.0
node 8 becomes faulty at time 200.0
node 9 becomes faulty at time 200.0
node 10 becomes faulty at time 200.0
node 11 becomes faulty at time 200.0
node 12 becomes faulty at time 200.0
node 13 becomes faulty at time 200.0
node 14 becomes faulty at time 200.0
node 15 becomes faulty at time 200.0
node 0 tests nodes 1 - 2 -> undefined at time 210.0.
node 0 tests nodes 4 - 8 -> undefined at time 210.0.
node 0 tests nodes 4 - 1 -> undefined at time 210.0.
node 0 tests nodes 4 - 2 -> undefined at time 210.0.
node 0 tests nodes 8 - 1 -> undefined at time 210.0.
node 0 tests nodes 8 - 2 -> undefined at time 210.0.
node 0 tests nodes 3 - 5 -> undefined at time 210.0.
node 0 tests nodes 3 - 1 -> undefined at time 210.0.
node 0 tests nodes 3 - 2 -> undefined at time 210.0.
node 0 tests nodes 3 - 4 -> undefined at time 210.0.
```

```
node 0 tests nodes 3 - 8 -> undefined at time 210.0.
node 0 tests nodes 5 - 1 -> undefined at time 210.0.
node 0 tests nodes 5 - 2 -> undefined at time 210.0.
node 0 tests nodes 5 - 4 -> undefined at time 210.0.
node 0 tests nodes 5 - 8 -> undefined at time 210.0.
node 0 tests nodes 6 - 9 -> undefined at time 210.0.
node 0 tests nodes 6 - 1 -> undefined at time 210.0.
node 0 tests nodes 6 - 2 -> undefined at time 210.0.
node 0 tests nodes 6 - 3 -> undefined at time 210.0.
node 0 tests nodes 6 - 4 -> undefined at time 210.0.
node 0 tests nodes 6 - 5 -> undefined at time 210.0.
node 0 tests nodes 6 - 8 -> undefined at time 210.0.
node 0 tests nodes 9 - 1 -> undefined at time 210.0.
node 0 tests nodes 9 - 2 -> undefined at time 210.0.
node 0 tests nodes 9 - 3 -> undefined at time 210.0.
node 0 tests nodes 9 - 4 -> undefined at time 210.0.
node 0 tests nodes 9 - 5 -> undefined at time 210.0.
node 0 tests nodes 9 - 8 -> undefined at time 210.0.
node 0 tests nodes 10 - 12 -> undefined at time 210.0.
node 0 tests nodes 10 - 1 -> undefined at time 210.0.
node 0 tests nodes 10 - 2 -> undefined at time 210.0.
node 0 tests nodes 10 - 3 -> undefined at time 210.0.
node 0 tests nodes 10 - 4 -> undefined at time 210.0.
node 0 tests nodes 10 - 5 -> undefined at time 210.0.
node 0 tests nodes 10 - 6 -> undefined at time 210.0.
node 0 tests nodes 10 - 8 -> undefined at time 210.0.
node 0 tests nodes 10 - 9 -> undefined at time 210.0.
node 0 tests nodes 12 - 1 -> undefined at time 210.0.
node 0 tests nodes 12 - 2 -> undefined at time 210.0.
node 0 tests nodes 12 - 3 -> undefined at time 210.0.
node 0 tests nodes 12 - 4 -> undefined at time 210.0.
node 0 tests nodes 12 - 5 -> undefined at time 210.0.
node 0 tests nodes 12 - 6 -> undefined at time 210.0.
node 0 tests nodes 12 - 8 -> undefined at time 210.0.
node 0 tests nodes 12 - 9 -> undefined at time 210.0.
node 0 tests nodes 7 - 11 -> undefined at time 210.0.
node 0 tests nodes 7 - 1 -> undefined at time 210.0.
node 0 tests nodes 7 - 2 -> undefined at time 210.0.
node 0 tests nodes 7 - 3 -> undefined at time 210.0.
node 0 tests nodes 7 - 4 -> undefined at time 210.0.
node 0 tests nodes 7 - 5 -> undefined at time 210.0.
node 0 tests nodes 7 - 6 -> undefined at time 210.0.
node 0 tests nodes 7 - 8 -> undefined at time 210.0.
node 0 tests nodes 7 - 9 -> undefined at time 210.0.
node 0 tests nodes 7 - 10 -> undefined at time 210.0.
node 0 tests nodes 7 - 12 -> undefined at time 210.0.
node 0 tests nodes 11 - 1 -> undefined at time 210.0.
node 0 tests nodes 11 - 2 -> undefined at time 210.0.
node 0 tests nodes 11 - 3 -> undefined at time 210.0.
node 0 tests nodes 11 - 4 -> undefined at time 210.0.
node 0 tests nodes 11 - 5 -> undefined at time 210.0.
node 0 tests nodes 11 - 6 -> undefined at time 210.0.
node 0 tests nodes 11 - 8 -> undefined at time 210.0.
node 0 tests nodes 11 - 9 -> undefined at time 210.0.
node 0 tests nodes 11 - 10 -> undefined at time 210.0.
node 0 tests nodes 11 - 12 -> undefined at time 210.0.
node 0 tests nodes 13 - 14 -> undefined at time 210.0.
node 0 tests nodes 13 - 1 -> undefined at time 210.0.
node 0 tests nodes 13 - 2 -> undefined at time 210.0.
node 0 tests nodes 13 - 3 -> undefined at time 210.0.
node 0 tests nodes 13 - 4 -> undefined at time 210.0.
node 0 tests nodes 13 - 5 -> undefined at time 210.0.
```

```
node 0 tests nodes 13 - 6 -> undefined at time 210.0.
node 0 tests nodes 13 - 7 -> undefined at time 210.0.
node 0 tests nodes 13 - 8 -> undefined at time 210.0.
node 0 tests nodes 13 - 9 -> undefined at time 210.0.
node 0 tests nodes 13 - 10 -> undefined at time 210.0.
node 0 tests nodes 13 - 11 -> undefined at time 210.0.
node 0 tests nodes 13 - 12 -> undefined at time 210.0.
node 0 tests nodes 14 - 1 -> undefined at time 210.0.
node 0 tests nodes 14 - 2 -> undefined at time 210.0.
node 0 tests nodes 14 - 3 -> undefined at time 210.0.
node 0 tests nodes 14 - 4 -> undefined at time 210.0.
node 0 tests nodes 14 - 5 -> undefined at time 210.0.
node 0 tests nodes 14 - 6 -> undefined at time 210.0.
node 0 tests nodes 14 - 7 -> undefined at time 210.0.
node 0 tests nodes 14 - 8 -> undefined at time 210.0.
node 0 tests nodes 14 - 9 -> undefined at time 210.0.
node 0 tests nodes 14 - 10 -> undefined at time 210.0.
node 0 tests nodes 14 - 11 -> undefined at time 210.0.
node 0 tests nodes 14 - 12 -> undefined at time 210.0.

Node 0 tests nodes 15 - 1 -> undefined at time 210.0.
Node 0 tests nodes 15 - 2 -> undefined at time 210.0.
Node 0 tests nodes 15 - 3 -> undefined at time 210.0.
Node 0 tests nodes 15 - 4 -> undefined at time 210.0.
Node 0 tests nodes 15 - 5 -> undefined at time 210.0.
Node 0 tests nodes 15 - 6 -> undefined at time 210.0.
Node 0 tests nodes 15 - 7 -> undefined at time 210.0.
Node 0 tests nodes 15 - 8 -> undefined at time 210.0.
Node 0 tests nodes 15 - 9 -> undefined at time 210.0.
Node 0 tests nodes 15 - 10 -> undefined at time 210.0.
Node 0 tests nodes 15 - 11 -> undefined at time 210.0.
Node 0 tests nodes 15 - 12 -> undefined at time 210.0.
Node 0 tests nodes 15 - 13 -> undefined at time 210.0.
Node 0 tests nodes 15 - 14 -> undefined at time 210.0.

Node 0 uses itself
Node 0 identifies through node 0 -> node 1 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 2 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 3 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 4 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 5 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 6 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 7 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 8 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 9 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 10 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 11 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 12 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 13 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 14 is Faulty at time 210.0.
Node 0 identifies through node 0 -> node 15 is Faulty at time 210.0.

Final Result:
Node 0 => TESTER
Node 1 Faulty at time 210.0.
Node 2 Faulty at time 210.0.
Node 3 Faulty at time 210.0.
Node 4 Faulty at time 210.0.
Node 5 Faulty at time 210.0.
Node 6 Faulty at time 210.0.
Node 7 Faulty at time 210.0.
Node 8 Faulty at time 210.0.
```

```
Node 9 Faulty at time 210.0.
Node 10 Faulty at time 210.0.
Node 11 Faulty at time 210.0.
Node 12 Faulty at time 210.0.
Node 13 Faulty at time 210.0.
Node 14 Faulty at time 210.0.
Node 15 Faulty at time 210.0.
```

Como pode-se constatar pelos dois experimentos, tanto com a existência de um único evento, como com a existência de $N - 1$ eventos simultâneos, o único nodo sem-falha do sistema consegue realizar o diagnóstico de todos os outros $N - 1$ nodos falhos. Portanto, o algoritmo é considerável $N - 1$-diagnosticável.

Capítulo 5

Conclusão

Neste trabalho apresentou-se um novo modelo para diagnóstico em nível de sistema baseado em comparações, bem como um novo algoritmo baseado nesse modelo, o algoritmo *Hi-Comp* (*Hierarchical Comparison-based Adaptive Distributed System-Level Diagnosis algorithm*), que é o primeiro algoritmo de diagnóstico ao mesmo tempo hierárquico, distribuído e baseado em comparações.

Algoritmos baseados em comparações executam o diagnóstico através da comparação de resultados de tarefas. No algoritmo *Hi-Comp* para os nodos realizarem o diagnóstico eles precisam testar outros nodos. Um teste é realizado através do envio de uma tarefa para dois nodos. Cada um desses nodos realiza a tarefa e devolve o resultado desta para o testador. Quando o testador recebe os dois resultados ele compara-os; se a comparação indicar

uma igualdade o testador considera que os dois nodos em questão estão sem-falha; já se a comparação indicar uma desigualdade o testador considera que pelo menos um dos dois nodos está falho, mas não consegue dizer qual deles.

Quando um nodo sem-falha testa outro nodo sem-falha, ele recebe do nodo testado informações de diagnóstico de todo o cluster ao qual o nodo testado pertence. Clusters são conjuntos de $N/2$ nodos. Como um nodo i pode receber informações de diagnóstico sobre um nodo j através de mais de um nodo, faz-se necessário garantir que o nodo i esteja sempre de posse da informação mais recente sobre o nodo j, para garantir que isso aconteça o algoritmo utiliza-se de *timestamps*, que são contadores de trocas de estados dos nodos.

O novo algoritmo realiza o diagnóstico de qualquer evento ocorrido no sistema em no máximo $log_2 N$ rodadas de testes, ou seja, a latência do algoritmo é de $log_2 N$ rodadas de testes. Uma rodada de testes é definida como o período de tempo que os nodos sem-falha do sistema levam para obter informações de diagnóstico sobre todos os nodos do sistema. Além disso, o algoritmo é $N-1$-diagnosticável, isto é, existindo no máximo $N-1$ nodos falhos no sistema o diagnóstico é possível.

O número máximo de testes no sistema pode chegar a $O(N^3)$ testes a cada rodada de testes. Porém, informalmente, especula-se que o número de testes não passe de $O(N^2)$, mas não existe nenhuma prova formal para este

limite.

A produção de uma ferramenta prática para gerência de falhas de redes de computadores baseada no algoritmo *Hi-Comp* é um dos principais objetivos de trabalhos futuros, bem como a investigação de um melhor limite para o número máximo de testes necessários.

Bibliografia

[1] G.S. Almasi, and A. Gottlieb, *Highly Parallel Computing,* The Benjamim/Cummings Publishing Company Inc., 1994.

[2] R.P. Bianchini, and R. Buskens, "Implementation of On-Line Distributed System-Level Diagnosis Theory," *IEEE Transactions on Computers*, Vol. 41, pp. 616-626, 1992.

[3] D.M. Blough, and H.W. Brown, "The Broadcast Comparison Model for On-Line Fault Diagnosis in Multicomputer Systems: Theory and Implementation," *IEEE Transactions on Computers*, Vol. 48, pp. 470-493, 1999.

[4] K.Y. Chwa, and S.L. Hakimi, "Schemes for Fault-Tolerant Computing: A Comparison of Modularly Redundant and t-Diagnosable Systems," *Information and Control*, Vol. 49, pp. 212-238, 1981.

[5] E.P. Duarte Jr., "SNMP-based Fault-Tolerant Network Monitoring," *Proc. XXV Seminário Integrado de Software e Hardware*, pp. 38-55, Belo Horizonte, 1998.

[6] E.P. Duarte Jr., "Um Algoritmo para Diagnóstico de Redes de Topologia Arbitrária," *Proc. I Workshop de Tolêrancia a Falhas*, pp. 50-55, Porto Alegre, 1998.

[7] E.P. Duarte Jr., A. Brawerman, and L.C.P. Albini, "A Diagnosis Algorithm based on Clusters with Detours," *disponível em http://www.inf.ufpr.br/˜elias.*

[8] E.P. Duarte Jr., A. Brawerman, and L.C.P. Albini, "An Algorithm for Distributed Hierarchical Diagnosis of Dynamic Fault and Repair Events," *Proc. IEEE ICPADS'00*, pp. 299-306, 2000.

[9] E.P. Duarte Jr., F. Mansfield, T. Nanya, and S. Noguchi, "Non-Broadcast Network Fault-Monitoring Based on System-Level Diagnosis," *Proc. IFIP/IEEE IM'97*, pp. 597-609, 1997.

[10] E.P. Duarte Jr., and T. Nanya, "A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm," *IEEE Transactions on Computers*, Vol.47, pp. 34-45, 1998.

[11] R. Gould, *Graph Theory,* The Benjamim/Cummings Publishing Company Inc., 1988.

[12] S.L. Hakimi, and A.T. Amin, "Characterization of Connection Assignments of Diagnosable Systems," *IEEE Transactions on Computers*, Vol. 23, pp. 86-88, 1974.

[13] S.L. Hakimi, and K. Nakajima, "On Adaptive System Diagnosis" *IEEE Transactions on Computers*, Vol. 33, pp. 234-240, 1984.

[14] F. Harary, *Graph Teory,* Addison-Wesley Publishing Company, 1971.

[15] S.H. Hosseini, J.G. Kuhl, and S.M. Reddy, "A Diagnosis Algorithm for Distributed Computing Systems with Failure and Repair," *IEEE Transactions on Computers*, Vol. 33, pp. 223-233, 1984.

[16] P. Jalote, *Fault Tolerance in Distributed Systems,* P T R Prentice Hall, 1994.

[17] S. Lee, and K.G. Shin, "Probabilistic Diagnosis of Multiprocessor Systems," *ACM Computing Surveys*, Vol. 26, No. 1, pp. 121-139, 1994.

[18] M.H. MacDougall, *Simulating Computer Systems: Techniques and Tools,* The MIT Press, Cambridge, MA, 1987.

[19] J. Maeng, and M. Malek, "A Comparison Connection Assigment for Self-Diagnosis of Multiprocessor Systems," *Digest 11th Int'l Symp. Fault Tolerant Computing*, pp. 173-175, 1981.

[20] M. Malek, "A Comparison Connection Assigment for Diagnosis of Multiprocessor Systems," *Proc. Seventh Int'l Symp. Computer Architecture*, pp. 31-36, 1980.

[21] F. Preparata, G. Metze, and R.T. Chien, "On The Connection Assignment Problem of Diagnosable Systems," *IEEE Transactions on Electronic Computers*, Vol. 16, pp. 848-854, 1968.

[22] S. Rangarajan, A.T. Dahbura, and E.A. Ziegler, "A Distributed System-Level Diagnosis for Arbitrary Network Topologies," *IEEE Transactions on Computers*, Vol. 44, pp. 312-333, 1995.

[23] A. Sengupta, and A.T. Dahbura, "On Self-Diagnosable Multiprocessor Systems: Diagnosis by Comparison Approach," *IEEE Transactions on Computers*, Vol. 41, No. 11, pp. 1386-1396, 1992.

[24] C. Xavier, and S.S. Iyengar, *Introduction to Parallel Algorithms,* Wiley-Intersciense Publication, 1998.

Apêndice A

Programa de Simulação

```
#include <stdlib.h>
#include <stdio.h>
#include "smpl.h"

/* events:  */

#define test   1
#define fault  2
#define repair 3

/* node description:  */

typedef struct {
  int id,       /* SMPL facility identifier */
    *tst,       /* TESTED-UP array */
    *scrap,     /* Undefined Nodes */
    cont,       /* number of tests */
    cluster,    /* cluster to test */
    node1,      /* nodes to be tested */
    node2,
    linha,      /* line wwith a fault-free node */
    all_faulty, /* indicates if exists any comparison ok */
    *tested,    /* nodes with info */
    *faulty,    /* Faulty nodes */
    *ff;        /* Fault-Free nodes */
} tnodo;

typedef struct lista{
  int info;
  struct lista *next;
}list;

tnodo *nodo;

static int N;  /* Number of Nodes passed likes parameter */
static int sabem=0, new_event=0,
  prev_round=0; /* variables to save the graphics file */


/*--- miscelaneous functions for Hi-ADSD ---*/

int elev (s)
```

```
     int s;
{  int  resp=1, i;
 for (i=1; i<=s; i++)
   resp = resp * 2;
 return resp;
}


int logtwo (s)
     int s;
     /* assumes input is a perfect power of two, greater than one */
{  int  resp=s/2, cont=1;
 while (resp != 1) { cont++; resp = resp/2; }
 return cont;
}


/*--- Cluster Find ---*/
unsigned int acha_cluster(unsigned int x,unsigned int y)
{
  unsigned int z,w,i;

  z=x^y;
  w=N/2;
  i=logtwo(N);
  while (((z&w)!=(N/2)) && (i!=0))
    {
      z=z<<1;
      i--;
    }
  return (i);
}


unsigned inverte(unsigned x)
{
  unsigned aux;
  int i;

  aux=0;
  for(i=1;i<=logtwo(N);i++)
      aux=(aux<<1)|1;

  return (aux-x);
}


/*--- return tester's sons ---*/
list *acha_filhos(unsigned tester)
{
  int son,i;
  list *sons, *aux, *sons_back;

  sons=(list *) malloc(sizeof(list));
  sons->info=0;
  sons->next=NULL;

  sons_back=sons;

  for(i=1;i<=logtwo(N);i++)
    {
      son=c_is(tester,i,0);
```

```
      aux=(list *) malloc(sizeof(list));
      aux->info=son;
      aux->next=NULL;
      sons->next=aux;
      sons=sons->next;
    }
  return sons_back->next;
}


/*--- return cluster's nodes ---*/
int c_is (i, s, j)
     int i, s, j;
{    int next, limite;
 limite = elev(s-1) - 1;
 if (i % elev(s) < elev(s-1))
   {     next = ((i % elev(s) + elev(s-1) + j) % elev(s)) +
           ((i / elev(s)) * elev(s));
   if (next < i) next = next + elev(s-1);
   }
 else
   next = ((i % elev(s) + elev(s-1) + j) % elev(s-1)) +
     ((i / elev(s)) * elev(s));
 return next;
} /* c_is */



/* --- returns a line of the tff_i hipercube ---*/
list *separa_linhas(int tester, int linha)
{
  list *sons, *aux, *sons_back;
  int *line;
  int i, cont;

  line = (int *) malloc (sizeof(int) * N);

  for (i=0;i<N;i++)
      line[i]=0;

  if ((linha<1) || (linha>logtwo(N))) /*---- test if the line exists ----*/
    return NULL;

  line[tester]=-1; /*---- remove the tester ----*/

  /*---- find the first line ----*/
  sons=acha_filhos(tester);
  while(sons!=NULL)
    {
      line[sons->info]=1;
      aux=sons->next;
      free(sons);
      sons=aux;
    }

  /*---- find which line nodes belong ----*/
  for (cont=1;cont<linha;cont++)
    {
      for (i=0;i<N;i++)
if (line[i]==cont)
  {
```

```
    sons=acha_filhos(i);
    while (sons!=NULL)
      {
if (line[sons->info]==0)
  line[sons->info]=cont+1;
aux=sons->next;
free(sons);
sons=aux;
      }
  }
    }

  /*---- separe the line for return ----*/
  aux=(list *) malloc(sizeof(list));
  aux->info=-1;
  aux->next=NULL;
  sons_back=aux;
  sons=aux;

  for(i=0;i<N;i++)
    {
      if (line[i]==linha)
{
  aux=(list *) malloc(sizeof(list));
  aux->info=i;
  aux->next=NULL;
  sons->next=aux;
  sons=sons->next;
}
    }

  free(line);
  return (sons_back->next);

}


/*--- find in which line a node is ---*/
int encontra_linha(int token, int node)
{
  list *linha,*linha_back;
  int ok=0,i;

  if (token == node)
    return 1;

  for (i=1; (i<=logtwo(N)) && (ok!=1);i++)
    {
      linha=separa_linhas(token,i);
      linha_back=linha;

      while (linha!=NULL)
{
  if (linha->info == node)
    ok = 1;
  linha=linha->next;
}

      while (linha_back!=NULL)
{
  linha=linha_back;
  linha_back=linha_back->next;
```

```
  free(linha);
}
    }
  return i;
}


/*--- choose the two nodes to be compared ---*/
void nodos_para_comparar(int token, int *node1, int *node2)
{
  int n1,n2,linha;
  list *lista1, *lista1_back, *aux;

  n1=*node1;
  n2=*node2;

  if (n1 == n2) /*---- first two sons of the tester ----*/
    {
      linha=1;
      lista1=separa_linhas(token,linha);
      lista1_back=lista1;
      *node1=lista1->info;
      *node2=lista1->next->info;
    }
  else
    {
      linha=encontra_linha(token,n2)-1;

      lista1=separa_linhas(token,linha);
      lista1_back=lista1;

      while (lista1!=NULL)
lista1=lista1->next;

      lista1=lista1_back;
      while (lista1->info!=n2)
lista1=lista1->next;

      if (lista1->next == NULL)  /*---- end of the line ---*/
{
  while(lista1_back->next != NULL)
    {
      aux=lista1_back;
      lista1_back=lista1_back->next;
      free(aux);
    }
  free(lista1_back);

  linha++;
  lista1=separa_linhas(token,linha);
  lista1_back=lista1;

  if (linha==logtwo(N)) /*---- last node ----*/
    {
      *node1=lista1->info;
      *node2=lista1->info;
    }
  else
    {
      *node1=lista1->info;
      *node2=lista1->next->info;
    }
```

```
}
      else
if (lista1->next->next == NULL) /*---- one node missing ----*/
  {
    *node1=*node2;
    *node2=lista1->next->info;
  }
else  /*---- at least two not tested nodes in the line ----*/
  {
    *node1=lista1->next->info;
    *node2=lista1->next->next->info;
  }
    }

  while(lista1_back->next != NULL)
    {
      aux=lista1_back;
      lista1_back=lista1_back->next;
      free(aux);
    }
  free(lista1_back);
}


/*--- send a task to two nodes ---*/
/*--- return 0 if the results match and 1 other wise ---*/
int send_task_and_compare(int token, int node1, int node2)
{
  int result;

  result = 1;
  if ((status(nodo[node1].id) == 0) && (status(nodo[node2].id) == 0))
    result = 0;

  return result;
}


/*--- tester obtains info from the tested fault-free node */
void more_info(unsigned tester, unsigned next_to_test)
{

  list *lista1, *lista1_back, *lista2, *lista2_back, *aux, *lst1;
  int i,linha;

  /*---- list with the sons of the fault-free node ----*/
  lista1=(list *) malloc(sizeof(list));
  lista1->info=-1;
  lista1->next=acha_filhos(next_to_test);
  lista1_back=lista1;
  lst1=lista1;
  /* --------------------------------------------- */

  lista2=(list *) malloc(sizeof(list));
  lista2->info=-1;
  lista2->next=NULL;
  lista2_back=lista2;

  linha=encontra_linha(tester,next_to_test);

  while(lista1_back->next!=NULL)
```

```
    {
      while(lista1->next!=NULL)
if(encontra_linha(tester,lista1->next->info) <= linha)
  {
    aux=lista1->next;
    lista1->next=lista1->next->next;
    free(aux);
  }
else
  lista1=lista1->next;

      if (lista1_back->next!=NULL)
{
  lista1_back=lista1_back->next;
  lista1->next=acha_filhos(lista1_back->info);
  linha=encontra_linha(tester,lista1_back->info);
}

      aux=(list *) malloc(sizeof(list));
      aux->info=lista1_back->info;
      aux->next=NULL;
      lista2->next=aux;
      lista2=lista2->next;
    }


  lista2=lista2_back->next;

  /*---- avoiding repetions ----*/
  aux=lista2;
  while (aux->next!=NULL)
    {
      lista2=aux->next;
      while (lista2->next!=NULL)
  if (lista2->next->info == aux->next->info)
    lista2->next=lista2->next->next;
  else
    lista2=lista2->next;
      if (aux->next!=NULL)
aux=aux->next;
    }
  lista2=lista2_back->next;

  /*---- obtains info about the cluster of the fault-free node ----*/
  while(lista2!=NULL)
    {
      printf("Node %d takes info about node %d from node %d -> ",tester,lista2->info,next_to_test);
      nodo[tester].tested[lista2->info]=1;

      if (nodo[next_to_test].tst[lista2->info] > nodo[tester].tst[lista2->info])
{
  nodo[tester].tst[lista2->info] = nodo[next_to_test].tst[lista2->info];
  nodo[tester].ff[lista2->info] = nodo[next_to_test].ff[lista2->info];
  nodo[tester].faulty[lista2->info] = nodo[next_to_test].faulty[lista2->info];

  if (nodo[tester].faulty[lista2->info] == 1)
    printf("Faulty.\n");
  else
    if (nodo[tester].ff[lista2->info] == 1)
      printf("Fault-Free.\n");
    else
      printf("\n");
```

```
}
      else
printf("\n");

      lista2=lista2->next;
    }

  while(lst1 != NULL)
    {
      aux=lst1;
      lst1=lst1->next;
      free(aux);
    }

  while (lista2_back != NULL)
    {
      aux=lista2_back;
      lista2_back=lista2_back->next;
      free(aux);
    }
}



/*--- send task to the undefined nodes and to the scrap ---*/
/*--- return 0 if finds a fault-free node and 1 other wise ---*/
int scrap_compare(int token, int node1, int node2)
{
  int i,k,
    result1=1,
    result2=1;

  for(i=0;i<N;i++)
    if (nodo[token].scrap[i]==1)
      {
result1=send_task_and_compare(token,node1,i);
if (result1==0) /*--- if finds any comparison ok ---*/
  {
    printf("node %d tests nodes %d - %d -> fault_free at time %5.1f.\n",token,node1,i,time());

    if (nodo[token].ff[i] != 1)
      nodo[token].tst[i]++;

    nodo[token].ff[i]=1;
    nodo[token].faulty[i]=0;
    nodo[token].scrap[i]=0;
    nodo[token].tested[i]=1;
    more_info(token,i);

    if (nodo[token].ff[node1] != 1)
      nodo[token].tst[node1]++;

    nodo[token].ff[node1]=1;
    nodo[token].faulty[node1]=0;
    nodo[token].tested[node1]=1;
    more_info(token,node1);

    if (nodo[token].faulty[node2] != 1)
      nodo[token].tst[node2]++;

    nodo[token].faulty[node2]=1;
    nodo[token].ff[node2]=0;
```

```
    nodo[token].tested[node2]=1;

    printf("Node %d identifies through node %d -> node %d is faulty at time %5.1f.\n"
                    ,token,node1,node2,time());

    for (k=0;k<N;k++)
      if (nodo[token].scrap[k]==1)
{
  printf("node %d identifies through node %d -> node %d is faulty at time %5.1f.\n"
                        ,token,node1,k,time());

  if (nodo[token].faulty[k] != 1)
    nodo[token].tst[k]++;

  nodo[token].faulty[k]=1;
  nodo[token].ff[k]=0;
  nodo[token].scrap[k]=0;
  nodo[token].tested[k]=1;
}
    break;
  }
else
  printf("node %d tests nodes %d - %d -> undefined at time %5.1f.\n",token,node1,i,time());
      }

  if (result1==1)
    {
      for(i=0;i<N;i++)
if (nodo[token].scrap[i]==1)
  {
    result2=send_task_and_compare(token,node2,i);
    if (result2==0) /*--- if finds any comparison ok ---*/
      {
printf("node %d tests nodes %d - %d -> fault_free at time %5.1f.\n"
                       ,token,node2,i,time());

if (nodo[token].ff[i] != 1)
  nodo[token].tst[i]++;

nodo[token].ff[i]=1;
nodo[token].faulty[i]=0;
nodo[token].scrap[i]=0;
nodo[token].tested[i]=1;
more_info(token,i);

if (nodo[token].ff[node2] != 1)
  nodo[token].tst[node2]++;

nodo[token].ff[node2]=1;
nodo[token].faulty[node2]=0;
nodo[token].tested[node2]=1;
more_info(token,node2);

if (nodo[token].faulty[node1] != 1)
  nodo[token].tst[node1]++;

nodo[token].faulty[node1]=1;
nodo[token].ff[node1]=0;
nodo[token].tested[node1]=1;

printf("Node %d identifies through node %d -> node %d is faulty at time %5.1f.\n"
                       ,token,node2,node1,time());
```

```
for (k=0;k<N;k++)
  if (nodo[token].scrap[k]==1)
    {
      printf("node %d identifies through node %d -> node %d is faulty at time %5.1f.\n"
                             ,token,node2,k,time());

      if (nodo[token].faulty[k] != 1)
nodo[token].tst[k]++;

      nodo[token].faulty[k]=1;
      nodo[token].ff[k]=0;
      nodo[token].scrap[k]=0;
      nodo[token].tested[k]=1;
    }
break;
      }
    else
      printf("node %d tests nodes %d - %d -> undefined at time %5.1f.\n",token,node2,i,time());
  }

      if (result2==1)  /*---- put nodes in the scrap */
{
  nodo[token].scrap[node1]=1;
  nodo[token].faulty[node1]=0;
  nodo[token].ff[node1]=0;
  nodo[token].tested[node1]=1;

  nodo[token].scrap[node2]=1;
  nodo[token].faulty[node2]=0;
  nodo[token].ff[node2]=0;
  nodo[token].tested[node2]=1;
}
    }

  if ((result1==0) || (result2==0))
    return 0;
  else
    return 1;
}


/*--- compares a fault-free node with the scrap ---*/
void compara_nodo_scrap(int token, int node)
{
  int i,j,result;

  for(i=0;i<N;i++)
    if (nodo[token].scrap[i]==1)
      {
result=send_task_and_compare(token,node,i);
if (result==0) /*---- finds a fault-free node, the rest of the scrap are faulty ----*/
  {
    printf("Node %d identifies through node %d -> node %d is Fault_free at time %5.1f.\n"
                   ,token,node,i,time());

    if (nodo[token].ff[i] != 1)
      nodo[token].tst[i]++;

    nodo[token].scrap[i]=0;
    nodo[token].ff[i]=1;
    nodo[token].faulty[i]=0;
```

```
    more_info(token,i);

    for(j=i;j<N;j++)
      if (nodo[token].scrap[j]==1)
{
  printf("node %d identifies through scrap from node %d -> node %d is Faulty at time %5.1f.\n"
                         ,token,node,j,time());

  if (nodo[token].faulty[j] != 1)
    nodo[token].tst[j]++;

  nodo[token].faulty[j]=1;
  nodo[token].ff[j]=0;
  nodo[token].scrap[j]=0;
}
  }
else
  {
    /*---- if node is faulty ----*/
    printf("Node %d identifies through node %d -> node %d is Faulty at time %5.1f.\n"
                   ,token,node,i,time());

    if (nodo[token].faulty[i] != 1)
      nodo[token].tst[i]++;

    nodo[token].scrap[i]=0;
    nodo[token].faulty[i]=1;
    nodo[token].ff[i]=0;
  }
      }
}


/*--- main loop ---*/
main(argc, argv)
     int  argc;
     char *argv[];

{
  static int event, /*---- event identifier, natural number ----*/
    token,          /*---- node identifier, natural number ----*/
    next_to_test,
    next_to_copy,
    i,j,
    r,
    ind, auxi;      /*---- auxiliar variables to make itoa ----*/

  static char fa_name[5];
  list *tests, *tests1, *tests2, *aux;

  int n,
    sair,
    k,
    linha,
    indefinido,
    result,
    node,
    acabei=0,
    ok;

  N = atoi(argv[1]);
```

```
   smpl(0,"Hi-ADSD Simulation");
   reset();
   stream(1);

   /*---- Inicialization ----*/

   nodo = (tnodo *) malloc(sizeof(tnodo)*N);

   for (i=0; i<N; i++) {
     nodo[i].cluster    = 1;
     nodo[i].cont       = 0;
     nodo[i].all_faulty = 1;
     nodo[i].node1      = i;
     nodo[i].node2      = i;
     nodo[i].linha      = logtwo(N);
     nodo[i].tst        = (int *) malloc (sizeof(int) * N);
     nodo[i].scrap      = (int *) malloc (sizeof(int) * N);
     nodo[i].faulty     = (int *) malloc (sizeof(int) * N);
     nodo[i].ff         = (int *) malloc (sizeof(int) * N);
     nodo[i].tested     = (int *) malloc (sizeof(int) * N);

     sprintf(fa_name, "%d", i);
     nodo[i].id         = facility(fa_name,1);

     for (j=0; j<N; j++)
       {
         nodo[i].tst[j]    = 0;
nodo[i].scrap[j]  = 0;
         nodo[i].ff[j]     = 0;
nodo[i].faulty[j] = 0;
nodo[i].tested[j] = 0;
       }
     nodo[i].tested[i]=1;
     nodos_para_comparar(i,&nodo[i].node1,&nodo[i].node2);
   }

   /*---- schedule initialization ----*/
   for (i=0; i<N; i++)
     schedule(test, 30.0, i);   /*---- i is token number ----*/

   while(time() < 400.0)
     {
       cause (&event, &token);

       switch(event) {

       case test:

if (status(nodo[token].id) != 0) break;

acabei=0;
sair=0;
ok=0;
while (sair!=1)
  {
    result=send_task_and_compare(token,nodo[token].node1,nodo[token].node2);
    if (result==0) /*---- test ok ----*/
      {
ok=1;

printf("Node %d tests nodes %d - %d -> fault_free at time %5.1f.\n"
       ,token,nodo[token].node1,nodo[token].node2,time());
```

```
if (nodo[token].ff[nodo[token].node1] != 1)
  nodo[token].tst[nodo[token].node1]++;

/*---- put nodes in ff ----*/
nodo[token].ff[nodo[token].node1]=1;
nodo[token].faulty[nodo[token].node1]=0;
nodo[token].tested[nodo[token].node1]=1;
more_info(token,nodo[token].node1);

if (nodo[token].ff[nodo[token].node2] != 1)
  nodo[token].tst[nodo[token].node2]++;

nodo[token].ff[nodo[token].node2]=1;
nodo[token].faulty[nodo[token].node2]=0;
nodo[token].tested[nodo[token].node2]=1;
more_info(token,nodo[token].node2);

/*---- compares with the scrap ----*/
compara_nodo_scrap(token,nodo[token].node1);

      }
    else    /*---- undefined test ----*/
      {
printf("node %d tests nodes %d - %d -> undefined at time %5.1f.\n"
       ,token,nodo[token].node1,nodo[token].node2,time());

/*---- compares with the scrap ----*/
result=scrap_compare(token,nodo[token].node1,nodo[token].node2);

if (result==0) /*---- comparison ok with the scrap ----*/
  ok=1;
      }

    if (ok==1)
      {
nodo[token].all_faulty=0;

nodo[token].linha=encontra_linha(token,nodo[token].node1);
      }

    nodos_para_comparar(token,&nodo[token].node1,&nodo[token].node2);

    if (encontra_linha(token,nodo[token].node1) > nodo[token].linha)
      {
nodo[token].node1=token;
nodo[token].node2=token;
nodos_para_comparar(token,&nodo[token].node1,&nodo[token].node2);
acabei = 1;
sair = 1;
      }
  }


if (nodo[token].all_faulty==1)
  {
    printf("\nLast Node\n");
    indefinido = 1;
    acabei = 1;
    /*---- last node, compares it with the scrap ----*/
    tests=separa_linhas(token,logtwo(N));
```

```
    for(i=0; ((i < N) && (indefinido != 0));i++)
      if (nodo[token].scrap[i]==1)
{
  result=send_task_and_compare(token,tests->info,i);
  if (result==0) /*---- find one fault-free node ----*/
    {
      indefinido=0;

      printf("Node %d tests nodes %d - %d -> Fault_free at time %5.1f.\n"
     ,token,tests->info,i,time());

      if (nodo[token].ff[i] != 1)
nodo[token].tst[i]++;

      nodo[token].scrap[i]=0;
      nodo[token].ff[i]=1;
      nodo[token].faulty[i]=0;
      nodo[token].tested[i]=1;
      more_info(token,i);

      if (nodo[token].ff[tests->info] != 1)
nodo[token].tst[tests->info]++;

      nodo[token].scrap[tests->info]=0;
      nodo[token].ff[tests->info]=1;
      nodo[token].faulty[tests->info]=0;
      nodo[token].tested[tests->info]=1;
      more_info(token,tests->info);

      for(j=i;j<N;j++)
if (nodo[token].scrap[j]==1)
  {
    printf("node %d identifies through node %d -> node %d is Faulty at time %5.1f.\n"
                                ,token,tests->info,j,time());

    if (nodo[token].faulty[j] != 1)
      nodo[token].tst[j]++;

    nodo[token].faulty[j]=1;
    nodo[token].ff[j]=0;
    nodo[token].scrap[j]=0;
  }
    }
  else
    {
      printf("Node %d tests nodes %d - %d -> undefined at time %5.1f.\n"
      ,token,tests->info,i,time());
    }
}

    if (indefinido == 1) /*---- put last node in the scrap ----*/
      nodo[token].scrap[tests->info]=1;
    else
      nodo[token].all_faulty=0;
    free(tests);
  }
else  /*---- tests node puts in teh scrap after a fault-free node was found ----*/
  if (acabei == 1)
    {
      /*---- takes a knowned fault-free node ----*/
      for(i=0;i<N;i++)
```

```
if (nodo[token].ff[i]==1)
  {
    node=i;
    break;
  }

      /*---- tests a scrap node with a fault-free node ----*/
      compara_nodo_scrap(token,node);

              /*---- test nodes with no information with a fault-free node ----*/
      for (i=0;i<N;i++)
if (nodo[token].tested[i] == 0)
  {
    result=send_task_and_compare(token,node,i);
    if (result==0) /*---- test ok ----*/
      {
printf("Node %d tests nodes %d - %d -> fault_free at time %5.1f.\n"
       ,token,node,i,time());

if (nodo[token].ff[i] != 1)
  nodo[token].tst[i]++;

nodo[token].ff[i]=1;
nodo[token].faulty[i]=0;
nodo[token].scrap[i]=0;
nodo[token].tested[i]=1;
more_info(token,i);
      }
    else
      {
printf("Node %d tests nodes %d - %d -> undefined. As node %d is fault-free => node %d is faulty at time %5.1f.\n"
                                ,token,node,i,node,i,time());

if (nodo[token].faulty[i] != 1)
  nodo[token].tst[i]++;

nodo[token].faulty[i]=1;
nodo[token].ff[i]=0;
nodo[token].scrap[i]=0;
nodo[token].tested[i]=1;
      }
  }
    }

/*---- compares the token with the scrap ----*/
if (nodo[token].all_faulty==1)
  {
    printf("\nItself\n");
    compara_nodo_scrap(token,token);
  }


if (acabei == 1)
  {
    /*---- Prints the final result ----*/
    printf("\n");
    for(i=0;i<N;i++)
      if (nodo[token].faulty[i]==1)
printf("Node %d Faulty at time %5.1f.\n",i,time());
      else
if (nodo[token].ff[i]==1)
  printf("Node %d Fault_Free at time %5.1f.\n",i,time());
```

```
else
  if (token == i)
    printf("Node %d => TESTER\n",i);

    printf("\n");

    /*---- reinicialization ----*/
    for (i=0;i<N;i++)
      if (i != token)
nodo[token].tested[i] = 0;
    nodo[token].all_faulty=1;
    nodo[token].linha=logtwo(N);
  }

schedule(test, 30.0, token);
printf("\n");

      break;

      case fault:
printf("node %d becomes faulty at time %5.1f\n", token, time());
r = request(nodo[token].id,token,0);
if (r != 0) {
  printf("Not possible to force fault");
  fflush(stdout);
  exit(0);
}
else
    schedule(repair, 1000.0, token);
break;

      case repair:
printf("node %d is repaired at time %5.1f\n", token, time());
release(nodo[token].id, token);
schedule(test, 0.0, token);   /* i is token number */
break;

      } /* end switch */
    } /* end while time */
}
```