ALESSANDRO BRAWERMAN

UMA ESTRATÉGIA DE TESTES ISÓCRONA PARA DIAGNÓSTICO HIERÁRQUICO DISTRIBUÍDO

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

Orientador: Prof. Elias P. Duarte Jr.

CURITIBA
2000

UFPR

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Comissão Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Alessandro Brawerman, avaliamos o trabalho intitulado **"Uma Estratégia de Testes Isócrona para Diagnóstico Hierárquico Distribuído"**, cuja defesa foi realizada no dia 19 de maio de 2000. Após a Avaliação, decidimos pela Aprovação do Candidato.

Curitiba, 19 de maio de 2000.

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

Profª. Dra. Ingrid Jansch Porto
Membro Externo - UFRS

Profª. Dra. Cristina Duarte Murta
Membro - DINF/UFPR

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

Agradecimentos

Agradeço a Deus por todas as grandes oportunidades que Ele me deu, inclusive a de estudar e me tornar o profissional que sou hoje em dia.

Agradeço a minha família pelo amor e carinho que sempre me deram, pelo apoio nas horas difíceis, por todos os preciosos conselhos dados a mim e por sempre serem meu porto seguro.

Agradeço ao meu orientador, Professor Elias Procópio Duarte Jr., por ser um grande orientador, por ter me dado conselhos que ajudaram muito na minha vida profissional e por ter me proporcionado a chance de realizar pesquisas científicas, uma das grandes paixões da minha vida. Elias, você é muito mais que um orientador, você é um amigo.

Agradeço ao amigo Luiz Carlos Pessoa Albini, que me persegue desde os tempos de faculdade, por ser este grande amigo. Luiz, juntos publicamos artigos, vencemos desafios, lutamos e sobrevivemos, espero que nossa amizade continue.

Agradeço ao amigo Aldri Luiz dos Santos, que conheci quando fui aceito no Mestrado, por ter sido o desbravador de certos softwares e nos ter passado tudo o que descobriu, por ter me ajudado em horas difícies e por todas as conversas e trocas de idéias que tivemos.

Agradeço a todos os amigos do Mestrado que de alguma forma, não só no ambiente profissional, me ajudaram a cada vez mais persistir e lutar por aquilo que eu acredito.

A todos, o meu muito Obrigado.

Conteúdo

Resumo

Diagnóstico em nível de sistema permite que todos os componentes de um sistema distribuído tolerante a falhas determinem quais outros componentes estão falhos e quais estão sem-falha. O período de tempo no qual os nodos que executam um determinado algoritmo de diagnóstico levam para completar o dignóstico de todo o sistema é chamado de latência do algoritmo.

Este trabalho apresenta um novo algoritmo distribuído para diagnóstico em nível de sistemas, o qual possui uma latência, no pior caso, de $O(logN)$ rodadas de testes para sistemas de N nodos. Algoritmos de diagnóstico hierárquico distribuído apresentados anteriormente, tais como Hi-ADSD e Hi-ADSD with Detours, possuem uma latência, no pior caso, de $O(log^2N)$ rodadas de testes.

O novo algoritmo, Iso Hi-ADSD, é baseado no algoritmo Hi-ADSD, agrupando os nodos em clusters para propósito de testes. Entretanto, o novo algoritmo emprega uma estratégia de testes isócrona na qual todos os nodos sem-falha executam testes em clusters de mesmo tamanho a cada rodada de testes. Esta estratégia é baseada em dois princípios: um nodo testado deve testar o cluster de seu testador na mesma rodada de testes; um nodo somente aceita testes de acordo com uma prioridade de ordem léxica.

Assume-se que uma rodada de testes é grande o suficiente para um nodo sem-falha testar e determinar o estado de até $N/2$ nodos. Todos os nodos sem-falha conhecem o início de cada rodada de testes. Provas formais do processo de tornar os testes isócronos e do pior caso da latência são apresentadas. São apresentados também resultados experimentais obtidos através de simulação.

Abstract

System-level diagnosis allows the components of a fault-tolerant distributed system to determine which components of the system are faulty and which are fault-free. The time it takes for nodes running the algorithm to diagnose a new event is called the algorithm's latency.

This work introduces a new distributed system-level diagnosis algorithm which presents a worst-case latency of $O(logN)$ testing rounds, for a system of N nodes. Some previous hierarchical distributed system-level diagnosis algorithms, Hi-ADSD and Hi-ADSD with Detours, presented a worst-case latency of $O(log^2N)$ testing rounds.

The new algorithm is based on Hi-ADSD, grouping nodes in progressively larger logical clusters for the purpose of testing. However, the new algorithm employs an isochronous testing strategy in which all fault-free nodes execute tests on clusters of the same size each testing round. This strategy is based on two main principles: a tested node must test its tester in the same testing round; a node only accepts tests according to a lexical priority order.

It is assumed that a testing round is large enough for a fault-free node to test and determine the state of up to $N/2$ nodes. Furthermore, all fault-free nodes know the beginning of a testing round. Formal proofs showing the proccess of tests becoming isochronous and the worst-case latency, are presented. Experimental results are also show through simulation.

Capítulo 1

Introdução

A popularização das redes de computadores e dos sistemas distribuídos tem permitido o desenvolvimento de novas aplicações de grande impacto para organizações. Ao mesmo tempo, estas organizações dependem cada vez mais do bom funcionamento de tais sistemas para realizarem suas tarefas. Desta forma, é fundamental o desenvolvimento de mecanismos para a construção de sistemas confiáveis.

Nos *sistemas distribuídos tolerantes a falhas* se um dos componentes falhar, o sistema como um todo não falha, isto é, continua desempenhando suas tarefas [1, 2]. Para construir um sistema tolerante a falhas é fundamental que se saiba quais componentes do sistema estão falhos. Os algoritmos de *Diagnóstico em Nível de Sistema*, introduzidos a seguir, permitem que os componentes sem-falha do sistema determinem quais componentes estão

falhos e quais componentes estão sem-falha.

1.1 Diagnóstico em Nível de Sistema

Considere um sistema composto de N nodos, os quais podem assumir um de dois estados: *falho* ou *sem-falha*. Um algoritmo de diagnóstico em nível de sistema permite a todos os nodos sem-falha obter informação de diagnóstico sobre o estado de todos os nodos no sistema [3].

Neste trabalho, assume-se que o grafo que representa o sistema é completo, isto é, existe um enlace de comunicação ligando qualquer par de nodos, e os enlaces não falham. Este modelo pode ser utilizado tanto para redes físicas, por exemplo, as redes locais baseadas em difusão, como para sistemas lógicos, por exemplo, um conjunto de computadores conectados à Internet.

Uma outra asserção comum aos algoritmos de diagnóstico em nível de sistema tratados neste trabalho é que os nodos falhos simplesmente deixam de funcionar, não respondendo aos testes aos quais são submetidos. Por outro lado, os nodos sem-falha são confiáveis, isto é, eles são capazes de executar testes e relatar seus resultados com precisão [4, 5]. Os testes são executados em intervalos pré-definidos, por exemplo, 5 milisegundos. Estes intervalos pré-definidos são chamados de *intervalos de testes*.

Alguns cuidados devem ser levados em conta na maneira de implemen-

tar os testes, de modo que os nodos testadores determinem corretamente se os nodos testados estão falhos ou sem-falhas. Uma estratégia simples na execução de testes consiste em considerar que os nodos testadores enviam estímulos aos nodos testados e aguardam respostas que devem chegar antes do intervalo de testes acabar. Se a resposta do nodo testado chega a tempo, o nodo testador conclui que testou um nodo sem-falha, caso contrário, o nodo testado é considerado falho. Para que um algoritmo obtenha sucesso na execução de sua estratégia de testes ele depende, frequentemente, da tecnologia na qual o sistema é baseado.

Um algoritmo de diagnóstico em nível de sistema é chamado *adaptativo* quando o conjunto de testes que um nodo executa é dinâmico, sendo escolhido a cada rodada de testes de acordo com os resultados de testes anteriores [6]. Os primeiros algoritmos de diagnóstico em nível de sistema assumiam a existência de um monitor, ou seja, um nodo que recebia o resultado de todos os testes executados e efetuava o diagnóstico. Por outro lado, os algoritmos que não assumem a existência do monitor, são chamados *distribuídos* [7]; neles, os próprios nodos que realizam os testes fazem o diagnóstico completo do sistema.

Dois critérios são geralmente utilizados para avaliar os algoritmos de diagnóstico em nível de sistema: o número de testes executados e a *latência*, isto é, o tempo necessário para que os nodos sem-falha completem o diagnóstico

do sistema.

O primeiro algoritmo ao mesmo tempo adaptativo *e* distribuído proposto foi o algoritmo Adaptive-DSD (*Adaptive Distributed System-Level Diagnosis*) [8, 9]. Aos nodos que executam este algoritmo são assinalados identificadores sequênciais, de 0 a $N-1$. Os nodos executam testes sequencialmente até encontrarem um nodo sem-falha. Quando um nodo sem-falha é testado, o nodo testador obtém as informações de diagnóstico que o nodo testado contém. A figura 1.1 mostra um exemplo de um sistema com 4 nodos executando o algoritmo Adaptive-DSD. Note que todos os nodos do sistema estão sem-falha e os testes recebidos por cada nodo sem-falha formam um grafo em anel.

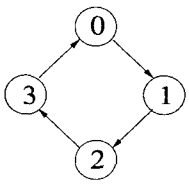

Figura 1.1: Sistema com 4 nodos executando o algoritmo Adaptive-DSD.

A latência e o número de testes do algoritmo Adaptive-DSD são definidos em termos de *rodadas de testes*. Uma rodada de testes é dita como o período de tempo no qual todos os nodos sem-falha do sistema testam um nodo sem-falha ou um nodo testa todos os outros nodos falhos. A latência do algoritmo é N rodadas de testes no pior caso. Por exemplo, considerando o sistema da figura 1.1, suponha que aconteça uma falha no nodo 3, o nodo 2

poderia demorar uma rodada de testes para diagnosticar esta falha. O nodo 1 poderia demorar mais uma rodada de testes para diagnosticar a falha do nodo 3 através do nodo 2, e, assim sucessivamente, até que todos os nodos do sistema completem o diagnóstico da falha no nodo 3. O número total de testes, considerando todos os nodos, é N a cada rodada de testes.

1.1.1 Diagnóstico Hierárquico

Em [10] foi proposto o algoritmo Hi-ADSD (*Hierarchical Adaptive Distributed System-Level Diagnosis*), no qual os nodos são agrupados em *clusters* para propósito de testes. Clusters são conjuntos de nodos. O número de nodos de um cluster, isto é, seu tamanho, é sempre uma potência de dois. Cada cluster possui um tamanho diferente podendo conter 1 nodo, 2 nodos, 4 nodos, ..., até $N/2$ nodos.

Neste algoritmo cada nodo executa seus testes começando pelo menor cluster, prosseguindo para o cluster com dois nodos, e assim por diante, até testar o maior cluster. Para obter informação de diagnóstico sobre um determinado cluster, o nodo testador deve executar testes naquele cluster até encontrar um nodo sem-falha, ou testar todos os nodos do cluster falhos. Se todos os nodos do cluster testado estiverem falhos, o testador passa a testar o próximo cluster. A figura 1.2, mostra um exemplo de um sistema com 4

nodos executando o algoritmo Hi-ADSD. Note que cada nodo por si só forma um cluster de tamanho 1 e a união de determinados pares de nodos forma clusters de tamanho 2. Neste exemplo, quando o nodo 1 testar o nodo 3 e este estiver sem-falha, o nodo 1 obterá informação de diagnóstico sobre todo o cluster do nodo 3, ou seja, sobre o nodo 2 e o nodo 3.

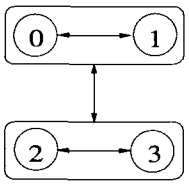

Figura 1.2: Um sistema com 4 nodos executando o algoritmo Hi-ADSD.

A latência do algoritmo Hi-ADSD é de, no máximo, log^2N rodadas de testes. Todos os logaritmos deste trabalho possuem base 2. Por exemplo, considere a figura 1.2. Suponha que ocorra uma falha no nodo 3, logo depois do nodo 2 tê-lo testado. O nodo 2 pode demorar, então, $logN$ rodadas de testes para diagnosticar a falha no nodo 3. O nodo 0 pode ainda, demorar mais $logN$ rodadas de testes para diagnosticar a falha do nodo 3 através do nodo 2. Generalizando, um nodo deve obter informações de diagnóstico de $logN$ clusters. Uma nova informação de diagnóstico pode levar até $logN$ rodadas de testes para ser propagada em um único cluster. Assim, para se propagar no sistema como um todo tal informação pode demorar até log^2N rodadas de testes. O número máximo de testes por rodada pode chegar a

$N^2/4$ testes.

Em [11] foi proposto o algoritmo *Hi-ADSD with Detours*, que utiliza a mesma estratégia de organizar os nodos em clusters introduzida pelo Hi-ADSD, mas possui uma estratégia de testes diferente. A cada rodada de testes um nodo testa um único cluster. Na primeira rodada, o cluster a ser testado possui tamanho igual a 1. Na segunda rodada, o cluster a ser testado possui tamanho igual a 2, na terceira rodada o cluster a ser testado possui tamanho igual a 4 e assim por diante, até o cluster de tamanho $N/2$ ser testado. Após testar este último cluster, o cluster de tamanho 1 é testado novamente, e repete-se todo o processo.

Quando um nodo testa um determinado cluster e o nodo testado estiver falho, o testador não precisa executar mais testes naquele cluster na mesma rodada de testes, pois ele poderá obter as informações de diagnóstico sobre o restante do cluster através de nodos sem-falha situados fora deste cluster, utilizando-se de caminhos alternativos chamados *detours*. Esta situação é ilustrada na figura 1.3, na qual o nodo 0 testa o nodo 2 falho e ao invés de testar o nodo 3, como acontece na estratégia de testes do algoritmo Hi-ADSD, o nodo 0 utiliza-se de um desvio (*detour*) através do nodo 1 para obter informação de diagnóstico do nodo 3. Nesta figura, a linha pontilhada corresponde ao teste que o nodo 0 evita ao utilizar o desvio.

A estratégia de testes do algoritmo Hi-ADSD with Detours garante que

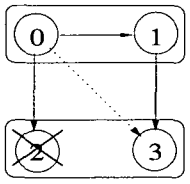

Figura 1.3: Nodos executando o algoritmo Hi-ADSD with Detours.

o número máximo de testes executados a cada $logN$ rodadas de testes é no máximo $NlogN$. Neste algoritmo a definição de rodada de testes é diferente da definição usada nos algoritmos anteriores. Uma rodada de testes, no algoritmo Hi-ADSD with Detours, é definida como o período de tempo no qual todos os nodos sem-falha realizam os testes necessários em um único cluster. A latência do algoritmo não é maior que log^2N rodadas de testes.

1.1.2 Uma Estratégia de Testes Isócrona

Nos algoritmos hierárquicos anteriores cada nodo pode estar testando um cluster de tamanho diferente a cada rodada de testes. Chamamos esta estratégia de testes de *não isócrona*. Neste trabalho é apresentado um novo algoritmo de diagnóstico hierárquico adaptativo e distribuído, o algoritmo *Iso Hi-ADSD*. O novo algoritmo apresenta uma latência máxima de $O(logN)$ rodadas de testes. Para permitir esta latência máxima o algoritmo emprega uma estratégia de testes que chamamos de estratégia de testes isócrona.

A estratégia de testes isócrona tem como principal característica o fato de que todos os nodos sem-falha testam clusters de mesmo tamanho em

cada rodada de testes. Para manter a estratégia de testes isócrona devemos utilizar dois princípios: um nodo testado deve testar o cluster de seu testador na mesma rodada de testes; um nodo somente aceita testes de acordo com uma prioridade de ordem léxica. Além disto, assume-se que uma rodada de testes é grande o suficiente para um nodo sem-falha testar e determinar o estado de até $N/2$ nodos. Todos os nodos sem-falha conhecem o início de cada rodada de testes.

São apresentadas provas formais que comprovam o processo de tornar os testes isócronos, ou seja, se os nodos de um sistema inicializam o algoritmo executando testes em clusters de tamanhos diferentes a cada rodada, em um número finito de rodadas de testes todos os nodos estarão executando testes em clusters de mesmo tamanho a cada rodada de testes. Além disto, prova-se que o número máximo de rodadas de testes para que todos os nodos realizem o diagnóstico do sistema é de $2logN - 1$. São apresentados também resultados experimentais obtidos através de simulação.

O restante deste trabalho está organizado da seguinte maneira. No capítulo 2 são apresentados os trabalhos relacionados. O capítulo 3 traz a especificação do novo algoritmo, Iso Hi-ADSD, bem como as provas formais. O capítulo 4 mostra resultados experimentais de simulações e considerações sobre a implementação prática do novo algoritmo. Por fim, a conclusão é apresentada no capítulo 5.

Capítulo 2

Diagnóstico em Nível de Sistema

A principal meta de um algoritmo de diagnóstico em nível de sistema é permitir que os nodos sem-falha descubram quais nodos do sistema estão falhos e quais estão sem-falha, isto é, os nodos sem-falha devem diagnosticar o estado de todos os outros nodos do sistema.

Neste capítulo é apresentado o modelo de diagnóstico PMC, bem como algoritmos de diagnóstico em nível de sistema baseados neste modelo.

2.1 O Modelo PMC

Todos os algoritmos apresentados neste trabalho seguem o modelo PMC, que foi proposto em 1968 por Preparata, Metze e Chien [4].

O modelo PMC considera um sistema composto por N nodos, os quais podem assumir um de dois estados: falho ou sem-falha. Os nodos do sis-

tema possuem a capacidade de executar testes em outros nodos e determinar se eles estão falhos ou sem-falha, ou seja, um nodo é capaz de determinar o estado de outros nodos através de testes. Um teste envolve a aplicação controlada de um estímulo e a observação da resposta correspondente. Os testes são executados em intervalos de tempo pré-definidos, por exemplo, 5 milisegundos. A implementação de um teste é dependente da tecnologia na qual o sistema é baseado.

O modelo PMC assume que os nodos sem-falha são confiáveis, isto é, eles são capazes de, após executarem seus testes, relatar os resultados com precisão. Por outro lado, os nodos falhos, após executarem seus testes, podem relatar resultados incorretos. A resposta de cada teste que um nodo testador executa em um outro nodo é representada por um bit, sendo 1 se o nodo testado está falho ou 0 se o nodo testado está sem-falha. O conjunto de todas as respostas dos testes de um sistema em um determinado intervalo, isto é, o conjunto de estados de todos os nodos do sistema, é chamado de síndrome do sistema.

O grafo que representa o sistema é completo, ou seja, existe um enlace de comunicação ligando qualquer par de nodos, e os enlaces não falham. Este modelo pode ser utilizado tanto para redes físicas, por exemplo, as redes locais baseadas em difusão, como para sistemas lógicos, por exemplo, um conjunto de computadores conectados à Internet.

A figura 2.1 ilustra um exemplo do modelo PMC representado por um sistema composto por 6 nodos. Suponha que somente um nodo, o nodo 0, está falho, então, a síndrome do sistema é representada da seguinte maneira: (X,0,0,0,0,1). Neste exemplo, o nodo 5 está sem-falha e por isso identifica corretamente o estado do nodo 0. Todos os outros nodos sem-falha também relatam seus resultados com precisão. O valor do resultado relatado pelo nodo 0, representado por X, pode ser tanto 0 como 1, pois este nodo está falho.

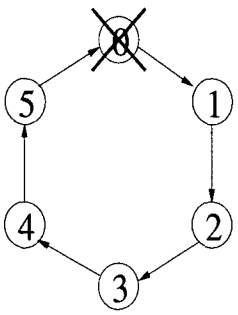

Figura 2.1: Modelo PMC, um sistema com 6 nodos.

Diferente do modelo PMC, os algoritmos apresentados neste trabalho levam em consideração falhas do tipo *crash*. Uma falha do tipo crash faz com que o nodo perca seu estado interno e simplesmente páre de funcionar. Com falhas deste tipo um nodo falho não executa testes e não responde aos testes a que é submetido.

2.1.1 Outros Modelos de Diagnóstico

Além do modelo PMC outros modelos de diagnóstico têm sido propostos. Os modelos conhecidos como *modelos de diagnóstico baseados em comparações*, como o próprio nome já diz, baseiam-se em comparações de resultados de testes para realizar o diagnóstico do sistema. Os principais modelos baseados em comparações são: o *Comparison-Based model* [12, 13], o modelo MM [14], o modelo de comparações generalizado [15] e o *Broadcast Comparison model* [16].

Outra abordagem alternativa é a dos modelos de diagnóstico probabilístico [17]. Neste modelo os nodos sem-falha possuem uma probabilidade x de realizar um diagnóstico correto do sistema.

2.2 Algoritmos Baseados no Modelo PMC

Nesta seção os algoritmos Adaptive-DSD (*Adaptive Distributed System-Level Diagnosis*), Hi-ADSD (*Hierarchical Adaptive Distributed System-Level Diagnosis*) e Hi-ADSD with Detours são apresentados. No próximo capítulo o novo algoritmo de diagnóstico distribuído hierárquico, o algoritmo Iso Hi-ADSD (Isochronous Hi-ADSD), é apresentado.

2.2.1 O Algoritmo Adaptive-DSD

Um algoritmo de diagnóstico em nível de sistema é chamado adaptativo quando o conjunto de testes que um nodo executa é dinâmico, sendo escolhido a cada rodada de testes de acordo com os resultados de testes anteriores [6]. Quando em um sistema não se assume a existência de um monitor, ou seja, os nodos que realizam os testes fazem o diagnóstico completo por si só, o algoritmo é dito distribuído [7].

O primeiro algoritmo de diagnóstico em nível de sistema ao mesmo tempo adaptativo *e* distribuído é o algoritmo Adaptive-DSD (*Adaptive Distributed System-Level Diagnosis*) [8, 9]. Aos nodos que executam o algoritmo são assinalados identificadores sequênciais, de 0 a $N-1$. Os nodos executam testes sequencialmente até encontrarem um nodo sem-falha, isto é, o nodo i testa o nodo $i+1$, se este estiver falho, o nodo i testa o nodo $i+2$ e assim por diante, desta forma o grafo de testes do sistema é um anel. A figura 2.2 mostra um sistema com 8 nodos executando o algoritmo Adaptive-DSD, note que os testes recebidos pelos nodos sem-falha do sistema formam um grafo de testes em anel.

Quando um nodo sem-falha é testado, o nodo testador obtém informações de diagnóstico do nodo testado. Estas informações incluem os resultados dos testes realizados pelo nodo testado, além das informações de diagnóstico

por ele obtidas, ou seja, ao testar um nodo sem-falha, o testador obtém informação de diagnóstico de todos os nodos do sistema.

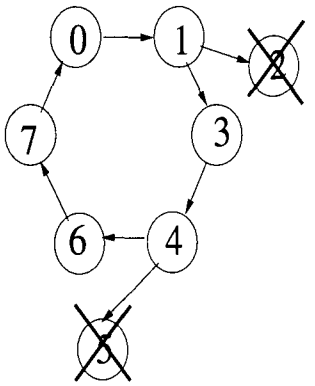

Figura 2.2: Nodos executando o algoritmo Adaptive-DSD.

Por outro lado, se um nodo i testado está falho, o testador procura o próximo nodo a ser testado, isto é, o testador testa o nodo $i+1$. Por exemplo, na figura 2.2 o nodo 1 testa o nodo 2 falho, então o nodo 1 procura o próximo nodo a ser testado, ou seja, o nodo 3. Como o nodo 3 está sem-falha, o nodo 1 então, obtém as informações de diagnóstico que o nodo 3 possui sobre o sistema.

A latência deste algoritmo é definida em termos de *rodadas de testes*. Uma rodada de testes é definida como o período de tempo no qual todos os nodos sem-falha do sistema testam um nodo sem-falha ou um nodo sem-falha testa todos os outros nodos falhos. O algoritmo Adaptive-DSD possui a propriedade de cada nodo ser testado exatamente uma vez a cada rodada

de testes e cada nodo sem-falha testar exatamente um outro nodo sem-falha por rodada de testes. Entretanto, se apenas um nodo estiver sem-falha e $N-1$ nodos estiverem falhos, o nodo sem-falha testará todos os $N-1$ nodos falhos. O número máximo de testes, considerando todos os nodos, é N a cada rodada de testes e a latência do algoritmo no pior caso é N rodadas de testes.

2.2.2 O Algoritmo Hi-ADSD

Em 1998 Duarte e Nanya [10] propuseram o algoritmo Hi-ADSD (*Hierarchical Adaptive Distributed System-Level Diagnosis*), que é hierárquico pois se baseia na estratégia de *dividir e conquistar* [18]. Neste algoritmo os nodos são agrupados em *clusters* para propósito de testes. Clusters são conjuntos de nodos. O número de nodos de um cluster, isto é, seu tamanho, é sempre uma potência de dois. Cada cluster possui um tamanho diferente podendo conter 1 nodo, 2 nodos, 4 nodos, ..., até $N/2$ nodos. O sistema como um todo é considerado um cluster com N nodos.

Um cluster de N nodos n_j, ..., n_{j+n-1}, onde j MOD $n = 0$, e n é uma potência de 2, é recursivamente definido como um nodo, no caso de $n = 1$; ou como a união de 2 clusters, um contendo os nodos n_j, ..., $n_{(j+n/2)-1}$ e o outro contendo nodos $n_{j+n/2}$, ..., n_{j+n-1}. A figura 2.3 mostra um sistema com 8

nodos organizados em clusters executando o algoritmo Hi-ADSD. Cada nodo é considerado um cluster de tamanho 1, a união de determinados pares de nodos é considerada um cluster de tamanho 2, a união de determinados 4 nodos um cluster de tamanho 4 e por fim a união dos 8 nodos um cluster de tamanho 8.

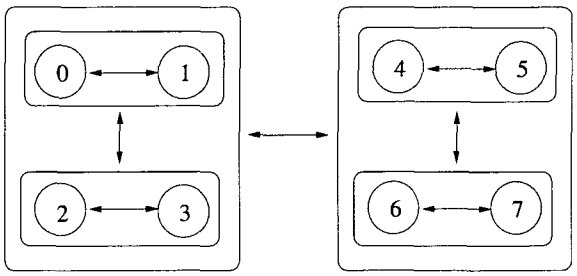

Figura 2.3: Um sistema com 8 nodos organizados em clusters.

No algoritmo Hi-ADSD cada nodo executa seus testes começando pelo menor cluster, prosseguindo para o cluster com dois nodos, e assim por diante, até testar o maior cluster. Este algoritmo emprega uma estratégia de testes *não isócrona*, ou seja, em uma determinada rodada de testes, cada nodo pode estar testando um cluster de tamanho diferente. Suponha que em uma rodada de testes x todos os nodos estão testando clusters de tamanho 1. Considere que na rodada de testes $x+1$ o nodo i falha, porém os outros nodos continuam testando clusters de mesmo tamanho. Quando o nodo i voltar a ficar sem-falha, neste algoritmo, não é garantido que o nodo recuperado, nodo i, execute testes em um cluster de mesmo tamanho que os outros nodos

sem-falha estão testando, caracterizando assim, uma estratégia de testes não isócrona

Para obter informação de diagnóstico sobre um determinado cluster, o nodo testador deve executar testes naquele cluster até encontrar um nodo sem-falha, ou testar todos os nodos do cluster falhos. Se um nodo sem-falha é encontrado, o testador obtém informação de diagnóstico sobre todo o cluster do nodo testado. Por exemplo, na figura 2.3, quando o nodo 0 testar o nodo 4 e este estiver sem-falha, o nodo 0 obterá informação de diagnóstico sobre todo o cluster do nodo 4, ou seja, sobre o nodo 4, o nodo 5, o nodo 6 e o nodo 7.

Seja um evento definido como a mudança do estado de um nodo, ou de falho para sem-falha ou de sem-falha para falho. A latência do algoritmo Hi-ADSD é de, no máximo, log^2N rodadas de testes.

A figura 2.4 mostra um exemplo de um sistema com 8 nodos que leva ao pior caso da latência do Hi-ADSD. Nesta figura, em uma determinada rodada de testes um evento ocorre no nodo 7. Devido ao fato da estratégia de teste empregada neste algoritmo ser não isócrona, o nodo 5 então, pode levar $logN$ rodadas de testes para diagnosticar o evento no nodo 7. O nodo 4 pode levar mais $logN$ rodadas de testes para diagnosticar o evento no nodo 7 a partir do nodo 5. O nodo 0 pode levar mais $logN$ rodadas de testes para diagnosticar o evento no nodo 7 a partir do nodo 4, o que dá um total de log^2N rodadas

de testes para o nodo 0 obter a nova informação de diagnóstico do nodo 7.

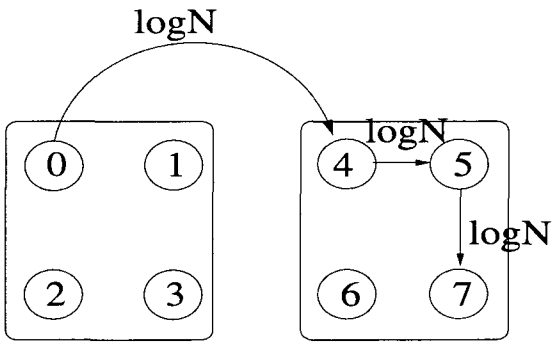

Figura 2.4: Pior caso da latência no algoritmo Hi-ADSD.

Considerando todos os nodos sem-falha do sistema, eles podem executar um total de $N^2/4$ testes em uma rodada de testes. Este pior caso ocorre, quando $N/2$ nodos estão falhos e pertencem a um único cluster, e os $N/2$ nodos sem-falha restantes, estão testando este cluster ao mesmo tesmpo.

A figura 2.5 mostra uma situação que desencadea o pior caso do número de testes. Nesta figura, o nodo 4, o nodo 5, o nodo 6 e o nodo 7 estão falhos e pertencem ao mesmo cluster. Os $N/2$ nodos sem-falha restantes estão executando testes no cluster que contém os nodos falhos ao mesmo tempo. Pela estratégia de testes do algoritmo Hi-ADSD, neste caso, o nodo 0 irá executar 4 testes, o nodo 1 mais 4 testes, o nodo 2 mais 4 testes e o nodo 3 mais 4 testes, o que leva a um total de 4 nodos * 4 testes= 16 testes, ou seja, $N^2/4$ testes em uma única rodada de testes.

O algoritmo Hi-ADSD é o primeiro algoritmo hierárquico de diagnóstico

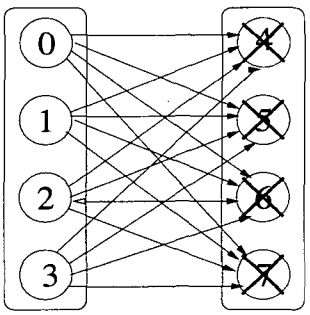

Figura 2.5: Pior caso do número de testes no algoritmo Hi-ADSD.

que é ao mesmo tempo adaptativo e distribuído. Outros algoritmos hierárquicos podem ser encontrados em [19, 20, 21, 22].

2.2.3 O Algoritmo Hi-ADSD with Detours

No trabalho proposto por Duarte, Brawerman e Albini [11], é apresentado o algoritmo *Hi-ADSD with Detours*, que utiliza a mesma estratégia de organizar os nodos em clusters introduzida pelo algoritmo Hi-ADSD, mas possui uma estratégia de testes diferente. A cada rodada de testes um nodo testa um único cluster. Na primeira rodada, o cluster a ser testado possui tamanho igual a 1, ou seja, possui apenas 1 nodo. Na segunda rodada, o cluster a ser testado possui tamanho igual a 2 , na terceira rodada o cluster a ser testado possui tamanho igual a 4 e assim por diante, até o cluster de tamanho $N/2$ ser testado. Após testar este cluster, o cluster de tamanho 1 é testado novamente, e repete-se todo o processo.

Neste algoritmo também não é garantido que todos os nodos sem-falha testem clusters de mesmo tamanho na mesma rodada de testes, ou seja, o algoritmo *Hi-ADSD with Detours* também emprega uma estratégia de testes não isócrona. No Hi-ADSD with Detours, quando um nodo testa um determinado cluster e o nodo testado estiver falho, o testador não precisa executar mais testes naquele cluster na mesma rodada de testes, pois ele poderá obter as informações de diagnóstico sobre o restante do cluster através de nodos sem-falha situados fora deste cluster, utilizando-se de caminhos alternativos chamados *detours*. Esta situação é mostrada na figura 2.6, na qual o nodo 0 testa o nodo 4 falho, então utiliza-se de desvios (*detours*) através do nodo 1, nodo 2 e nodo 3 para obter, respectivamente, informação do nodo 5, nodo 6 e nodo 7. Nesta figura, as linhas pontilhadas correspondem aos testes que o nodo 0 evita ao utilizar desvios.

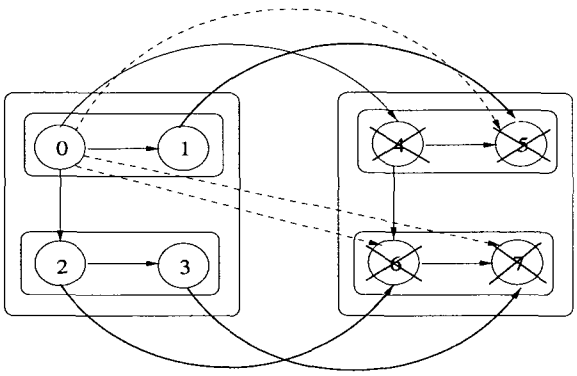

Figura 2.6: Nodos executando o algoritmo Hi-ADSD with Detours.

A estratégia de testes do algoritmo Hi-ADSD with Detours garante que

o número máximo de testes executados a cada $logN$ rodadas de testes é no máximo $NlogN$. Neste algoritmo uma rodada de testes é definida como o período de tempo no qual todos os nodos sem-falha realizam os testes necessários em um único cluster. A latência do algoritmo não é maior que log^2N rodadas, o mesmo pior caso do algoritmo Hi-ADSD.

2.2.4 Comparação entre os Algoritmos Apresentados

Dois critérios são geralmente utilizados para avaliar os algoritmos de diagnóstico em nível de sistema: o número de testes executados e a latência, isto é, o tempo total para que todos os nodos sem-falha completem um diagnóstico do sistema.

Algoritmos	*ADSD*	*Hi-ADSD*	*Hi-ADSD with Detours*
Número total de testes (pior caso)	N por rodada de teste	$N^2/4$ por rodada de teste	$NlogN$ por $logN$ rodadas de teste
Latência (pior caso)	N rodadas de testes	log^2N rodadas de testes	log^2N rodadas de testes

Tabela 2.1: Comparação dos algoritmos de diagnóstico.

A tabela 2.1 mostra os dois critérios de avaliação acima citados nos principais algoritmos que influenciaram este trabalho. Podemos notar que:

(1) Quanto ao número total de testes no pior caso, o algoritmo Hi-ADSD é o que apresenta o maior número de testes em uma mesma rodada. Os

algoritmos Adaptive-DSD e Hi-ADSD with Detours apresentam números de testes semelhantes: no primeiro, o máximo é de N testes por rodada de testes e, no segundo, o máximo é de $NlogN$ testes a cada $logN$ rodadas de testes.

(2) Quanto à latência no pior caso, isto é, o tempo necessário para que os nodos sem-falha completem o diagnóstico do sistema, o algoritmo Adaptive-DSD é o que apresenta o maior tempo para completar o diagnóstico: uma latência, no pior caso, de no máximo N rodadas de testes. Os algoritmos Hi-ADSD e Hi-ADSD with Detours apresentam latência no pior caso de no máximo log^2N rodadas de testes.

Devido ao fato de que os algoritmos anteriores empregam uma estratégia de testes não isócrona, a latência é de, no máximo, log^2N rodadas de testes. No novo algoritmo usamos uma estratégia de testes chamada *isócrona*, isto é, todos os nodos sem-falha testam clusters de mesmo tamanho a cada rodada de testes, que nos permite obter uma latência máxima de, O($logN$) rodadas de testes.

No próximo capítulo o algoritmo Iso Hi-ADSD é especificado e são apresentadas provas formais do processo de tornar os testes isócronos e do pior caso da latência.

Capítulo 3

Um Novo Algoritmo para Diagnóstico Hierárquico

Neste capítulo é apresentada a especificação do novo algoritmo, Iso Hi-ADSD. Também são apresentadas as provas formais do processo de tornar os testes isócronos e da latência máxima.

3.1 O Algoritmo Iso Hi-ADSD: Especificação

Considere um sistema S, composto por N nodos, n_0, n_1, ..., n_{N-1}. Neste trabalho nos referimos ao nodo n_i como nodo i. Assuma que o grafo que representa o sistema é completo, isto é, existe um enlace de comunicação entre todos os pares de nodos (n_i, n_j) e os enlaces não falham. Cada nodo n_i pode assumir um de dois estados: falho ou sem-falha. A coleção de estados de todos os nodos do sistema, em um determinado intervalo, é chamada de síndrome do sistema.

Como no modelo PMC, neste novo algoritmo assume-se que os nodos

sem-falha são confiáveis, isto é, eles são capazes de executar testes e relatar seus resultados com precisão. Porém, de forma diferente do modelo PMC, o algoritmo Iso Hi-ADSD, leva em consideração falhas do tipo crash, ou seja, os nodos falhos simplesmente deixam de funcionar, não respondendo aos testes aos quais são submetidos. Esta é uma asserção comum em algoritmos práticos de diagnóstico [8, 9, 10, 11, 23].

Nodos executam testes em outros nodos em intervalos de tempo pré-definidos, por exemplo, 5 milisegundos. Um teste envolve a aplicação controlada de um estímulo e a observação da resposta correspondente. A implementação de um teste é dependente da tecnologia na qual o sistema é baseado.

O novo algoritmo utiliza a estratégia de agrupar nodos em clusters lógicos para propósito de testes, estratégia esta definida no algoritmo Hi-ADSD [10]. Cluster são conjunto de nodos. O número de nodos em um cluster determina o tamanho do cluster, e este número é sempre uma potência de 2. Inicialmente, N é assumido ser uma potência de 2, e o sistema inteiro é um cluster de N nodos.

Um cluster de N nodos n_j, ..., n_{j+n-1}, onde j MOD $n = 0$, e n é uma potência de 2, é recursivamente definido como um nodo, no caso de $n = 1$; ou como a união de dois clusters, um contendo os nodos n_j, ..., $n_{(j+n/2)-1}$ e o outro contendo nodos $n_{j+n/2}$, ..., n_{j+n-1}. A figura 3.1 mostra um sistema

com 8 nodos organizados em clusters. Cada nodo é considerado um cluster de tamanho 1, a união de determinados pares de nodos é considerada um cluster de tamanho 2, a união de determinados 4 nodos, um cluster de tamanho 4; e, por fim, a união dos 8 nodos um cluster de tamanho 8, contendo todos os nodos do sistema.

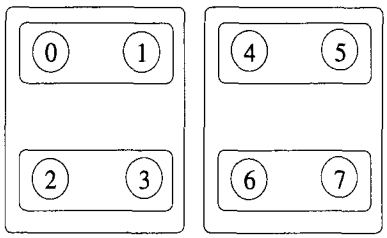

Figura 3.1: Nodos agrupados em clusters.

Na primeira rodada de testes, cada nodo sem-falha executa seus testes em nodos que pertencem a clusters de tamanho 1. Na segunda rodada, os nodos sem-falha executam testes em nodos que pertencem a clusters de tamanho 2. Na terceira rodada, testes são executados em nodos que pertencem a clusters de tamanho 4 e assim por diante até o cluster de tamanho 2^{logN-1}, isto é, o cluster de tamanho $N/2$ ser testado. Após este último teste, os nodos que pertencem ao cluster de tamanho 1 são testados novamente e repete-se todo o processo.

Seja o *grafo de testes livre de falhas de um sistema*, $TFF(S)$, definido como um grafo direcionado no qual os vértices são nodos do sistema S. Existe

uma aresta do nodo a para o nodo b, se o nodo a testa o nodo b sem-falha no mais recente intervalo de testes. Quando todos os nodos do sistema estão sem-falha, $TFF(S)$ tem a forma de um hipercubo. A figura 3.2 exemplifica um grafo $TFF(S)$ para um sistema de 8 nodos. Note que, por exemplo, se o nodo 0 testa o nodo 1 sem-falha, então existe uma aresta que liga o nodo 0 ao nodo 1. Por sua vez, se o nodo 1 testa o nodo 0 sem-falha, então existe uma aresta que liga o nodo 1 ao nodo 0.

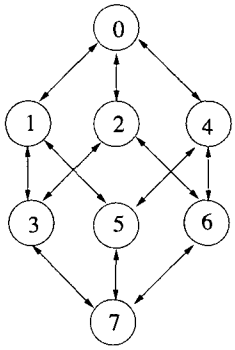

Figura 3.2: $TFF(S)$ para um sistema de 8 nodos.

Seja um *caminho direcionado* partindo do nodo i até o nodo j, definido como uma lista de nodos, que começa no nodo i e termina no nodo j, na qual cada nodo possui uma aresta direcionada para o próximo nodo. O caminho direcionado do nodo i ao nodo j é a rota através da qual a informação de diagnóstico sobre o nodo j deve passar até atingir o nodo i. Por exemplo, na figura 3.2 um caminho direcionado entre o nodo 0 e o nodo 7 é formado pela rota (nodo 0, nodo 1, nodo 3 e nodo 7).

Seja a *distância de diagnóstico* entre o nodo i e o nodo j, definida como o número de nodos, através dos quais a informação de diagnóstico sobre o nodo j deve passar até atingir o nodo i, incluindo o próprio nodo j. Por exemplo, na figura 3.2 a distância de diagnóstico entre o nodo 0 e o nodo 7 é igual a 3.

Um *grafo de testes livre de falhas do nodo* i, TFF_i, é definido como um grafo direcionado no qual os nodos são nodos do sistema S e existe um caminho direcionado do nodo a para o nodo b, se o nodo a testa o nodo b e a distância de diagnóstico do nodo i para o nodo a é menor do que a distância de diagnóstico do nodo i para o nodo b em $TFF(S)$. A figura 3.3 exemplifica TFF_0 para um sistema com 8 nodos. Note que, existe um caminho direcionado do nodo 0 ao nodo 3, passando pelo nodo 1, pois a distância de diagnóstico do nodo 0 ao nodo 3, em $TFF(S)$, é igual a dois e a distância de diagnóstico do nodo 1 ao nodo 3, em $TFF(S)$, é igual a 1. Então, obrigatoriamente, o caminho direcionado entre o nodo 0 e o nodo 3, deve passar pelo nodo 1.

A lista ordenada de nodos que podem ser atingidos pelo nodo i, em TFF_i, com distância de diagnóstico menor ou igual a s, é chamada $c_{i,s}$. Na verdade a lista $c_{i,s}$ é uma função definida no algoritmo Hi-ADSD que tem como objetivo montar os diversos clusters do sistema e determinar para o testador qual nodo deve ser testado em cada intervalo de testes.

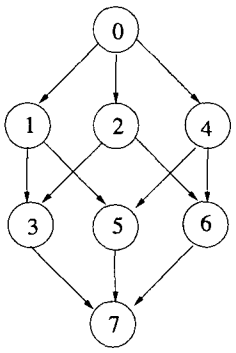

Figura 3.3: TFF_0 para um sistema com 8 nodos.

Conforme a estratégia de testes, a cada rodada de testes, o nodo i executa testes nos nodos pertencentes a uma determinada $c_{i,s}$. Na primeira rodada de testes s é igual a 0. Na segunda rodada de testes s é igual a 1, e assim por diante, até s atingir o valor de $logN$. Após o nodo i ter testado $c_{i,logN}$, repete-se todo o processo.

Quando um nodo i testa uma determinada $c_{i,s}$, ele executa testes sequencialmente até encontrar um nodo sem-falha ou testar todos os nodos pertencentes àquela $c_{i,s}$ como falhos. Supondo que um nodo sem-falha seja encontrado; a partir deste nodo sem-falha o nodo i copia a informação de diagnóstico de todos os nodos pertencentes àquela $c_{i,s}$. Para um sistema representado na figura 3.2 a tabela 3.1 lista toda $c_{i,s}$ para i variando de 0 a 7 e para s variando de 1 a 3.

Seja uma rodada de testes definida como o período de tempo em que todos os nodos sem-falha executam um conjunto de intervalos de testes previamente

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2,3	3,2	0,1	1,0	6,7	7,6	4,5	5,4
3	4,5,6,7	5,6,7,4	6,7,4,5	7,4,5,6	0,1,2,3	1,2,3,0	2,3,0,1	3,0,1,2

Tabela 3.1: Listagem de toda e qualquer $c_{i,s}$ para um sistema de 8 nodos.

definidos em um único cluster. A figura 3.4, que representa o grafo TFF_0, mostra três rodadas de testes consecutivas nas quais o nodo 0 testa em cada uma delas uma $c_{0,s}$ diferente. Na primeira rodada de testes o nodo 0 testa o cluster de tamanho 1, ou seja, a $c_{0,1}$. Ao testar a $c_{0,1}$, o nodo 0 copia a informação de diagnóstico do nodo 1 e acaba-se a primeira rodada de testes.

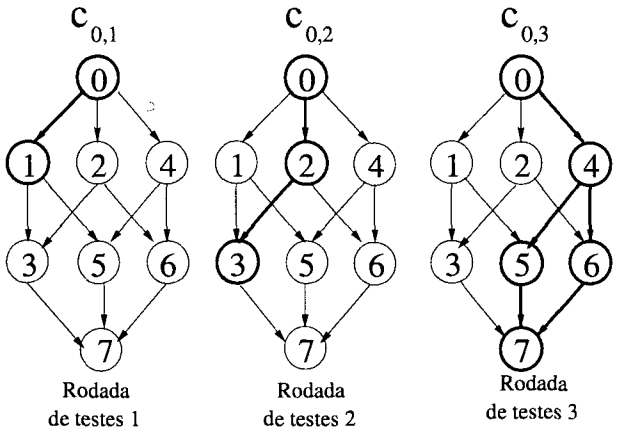

Figura 3.4: Exemplo de $c_{0,s}$ em três rodadas de testes diferentes.

Na segunda rodada de testes o nodo 0 testa o cluster de tamanho 2, ou seja, a $c_{0,2}$. Ao testar a $c_{0,2}$, o nodo 0 copia as informações de diagnóstico do nodo 2. Estas informações contém o estado atual do nodo 2 e as informações que o nodo 2 possui a respeito do cluster ao qual ele pertence, neste caso,

informações a respeito do nodo 3. Após este teste, acaba-se a segunda rodada de testes.

Na terceira rodada de testes o nodo 0 testa o cluster de tamanho 4, ou seja, a $c_{0,3}$. Ao testar a $c_{0,3}$, o nodo 0 copia as informações de diagnóstico do nodo 4. Estas informações contém o estado atual do nodo 4 e as informações que o nodo 4 possui a respeito do cluster ao qual ele pertence, neste caso, informações a respeito do nodo 5, nodo 6 e nodo 7. Após este teste, acaba-se a terceira rodada de testes e repete-se todo o processo.

Uma característica deste novo algoritmo é que todos os nodos sem-falha do sistema ou testam somente um nodo sem-falha em outro cluster por rodada de testes, ou testam todos os nodos daquele cluster como falhos em uma única rodada de testes.

3.1.1 Estratégia de Testes Isócrona

O novo algoritmo tem por objetivo diminuir a latência de diagnóstico no seu pior caso. Para tal é proposta a utilização de uma estratégia de testes isócrona, isto é, uma estratégia na qual todos os nodos sem-falha do sistema testam clusters de mesmo tamanho em cada rodada de testes. Devido ao fato que nos algoritmos anteriores não existe tal estratégia, ou seja, eles empregam uma estratégia de testes não isócrona, isto é, uma estratégia na qual os nodos

sem-falha do sistema podem testar clusters de tamanho diferente em cada rodada de testes, a latência máxima obtida é de log^2N rodadas de testes [10, 11].

A figura 3.5 mostra um sistema no qual os testes são executados de maneira não isócrona. Por exemplo, o nodo 0 e o nodo 7 estão testando um cluster de tamanho 4, enquanto que o nodo 1 e o nodo 2 estão testanto um cluster de tamanho 1.

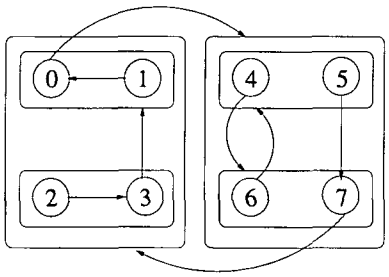

Figura 3.5: Testes executados de maneira não isócrona.

A figura 3.6 mostra um sistema no qual os testes são executados de maneira isócrona. Nesta figura todos os nodos estão sem-falha e realizam testes em clusters de tamanho 2.

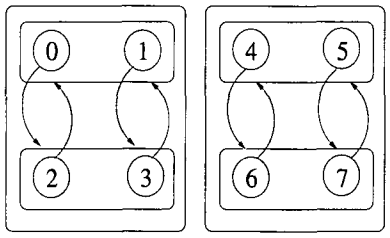

Figura 3.6: Testes executados de maneira isócrona.

De modo a garantir a estratégia de testes isócrona são assumidas duas asserções no momento da execução do algoritmo Iso Hi-ADSD:

1. Todos os nodos sem-falha conhecem o início de cada rodada de testes.

2. Uma rodada de testes é grande o suficiente para um nodo sem-falha testar e determinar o estado de até $N/2$ nodos.

A primeira asserção é necessária para determinarmos corretamente o início de cada rodada de testes. Para a implementação desta asserção, no início de cada rodada de testes, uma mensagem pode ser difundida para todos os nodos sem-falha. Quando os nodos sem-falha recebem a mensagem, eles começam a executar os testes. Uma ferramenta de difusão confiável pode ser empregada para este propósito [16].

A segunda asserção é necessária pois, dependendo do número de nodos falhos contidos em um cluster, alguns clusters podem demorar mais tempo para serem testados. Como o maior cluster de um sistema possui $N/2$ nodos, um nodo sem-falha poderá precisar de $N/2$ intervalos de testes, no pior caso, para testar o cluster inteiro e terminar a rodada de testes atual. Em outras palavras, esta asserção determina o tempo máximo que deve-se esperar para que se inicie uma outra rodada de testes.

A figura 3.7 mostra como as duas asserções, quando usadas em conjunto, facilitam o processo de tornar os testes isócronos. Nesta figura três nodos

executam testes em um sistema S qualquer. Em 3.7-A as rodadas de testes começam e terminam em instantes de tempo diferentes para nodos diferentes. Neste exemplo, as duas asserções não são respeitadas, sendo assim, o processo de tornar os testes isócronos é comprometido. Em 3.7-B as rodadas de testes começam e terminam exatamente no mesmo instante de tempo para todos os nodos sem-falha. Neste exemplo, as duas asserções são respeitadas, tornando possível aos nodos determinarem o início de cada rodada de testes.

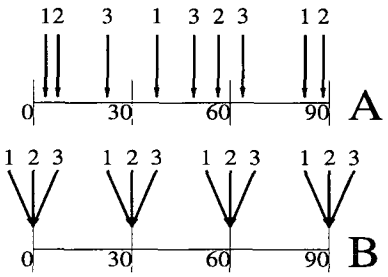

Figura 3.7: Delimitando e início e o fim das rodadas de testes.

3.2 Estratégia de Testes do Novo Algoritmo

Tornar a execução dos testes isócrona significa que, se um nodo do sistema S sem-falha está testando um cluster com x nodos, ou seja, um cluster de tamanho x, então, todos os nodos sem-falha do sistema S deverão estar testando um cluster de tamanho x numa mesma rodada de testes. Considere o nodo i e o nodo k ambos sem-falha pertencentes ao sistema S, se o nodo i

está testando um cluster com 4 nodos, então o nodo k deve testar um cluster com 4 nodos na mesma rodada de testes.

Em nosso sistema S, a estratégia de testes isócrona deve ser garantida mesmo se ocorrerem mudanças de estado nos nodos da rede, ou seja, se um nodo falho voltar a ficar sem-falha ele deverá testar um cluster do mesmo tamanho que os outros nodos estão testando. A figura 3.8 mostra uma situação em que na primeira rodada de testes todos os nodos estão testando clusters de mesmo tamanho, então na rodada 2 o nodo 3 falha, porém os testes continuam isócronos. Na rodada 3, o nodo 3 volta ao estado sem-falha e deverá recomeçar todo o processo, e assim, conseguir tornar seus testes isócronos com os dos outros nodos sem-falha.

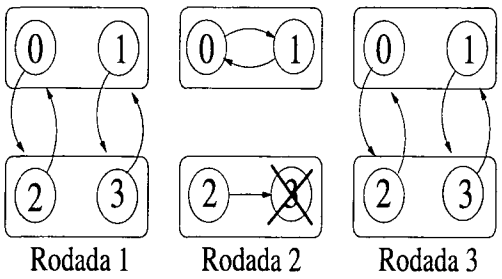

Figura 3.8: Um sistema S executando o novo algoritmo.

O algoritmo Iso Hi-ADSD é baseado em dois princípios básicos para alcançar e manter os testes isócronos.

Princípio 1

Se um nodo x testa um nodo y, sem-falha, pertencente a $c_{x,k}$ então o nodo y deve testar $c_{y,k}$, ao qual o nodo x pertence, na mesma rodada de testes.

Se existir algum nodo y sem-falha no sistema que é testado por um nodo x em $c_{x,k}$, e este nodo não executa um teste em $c_{y,k}$ na mesma rodada de testes, então o sistema possui uma estratégia de testes não isócrona.

A figura 3.9 mostra um sistema S com 4 nodos e uma estratégia de testes não isócrona. Neste exemplo o nodo 0, o nodo 2 e o nodo 3 estão executando testes de forma isócrona, ou seja, estes nodos estão testando clusters de mesmo tamanho, porém o nodo 1 é testado pelo nodo 0 e o nodo 1 não executa um teste no cluster ao qual o nodo 0 pertence na mesma rodada de testes. Este teste que o nodo 1 deixa de executar, no cluster do nodo 0, caracteriza a estratégia de testes do algoritmo como não isócrona.

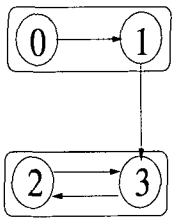

Figura 3.9: Um sistema S com uma estratégia de testes não isócrona.

Princípio 2

É necessário um esquema de prioridade no momento de tornar os testes isócronos.

Um esquema de prioridade é usado pelos nodos para garantir a estratégia de testes isócrona. Se um nodo com maior prioridade testa um nodo com menor prioridade, então o nodo testado deve executar testes no cluster ao qual o nodo testador pertence na mesma rodada de testes. Por outro lado, se o testador possui menor prioridade, o nodo testado recusa o teste enviando uma mensagem ao testador informando em qual cluster o nodo testado está executando testes nesta rodada.

A figura 3.10 mostra uma situação em que na primeira rodada de testes os nodos executam testes de forma isócrona, na segunda rodada o nodo 3 falha, porém os testes continuam isócronos. Já na terceira rodada o nodo 3 volta a ficar sem-falha e executa um teste no nodo 2, o qual possui testes isócronos com os testes dos outros nodos e por isto esta recebendo um teste do nodo 0. Neste momento o nodo 2 deve decidir se responde ao nodo 0 e continua com seus testes isócronos com todos os outros nodos, ou se responde ao nodo 3 e deixa de executar testes isócronos com os outros nodos.

O esquema de prioridade no processo de tornar os testes isócronos que será adotado neste novo algoritmo é baseado na prioridade léxica. Cada nodo guarda em uma variável, chamada `iso`, o identificador do nodo de

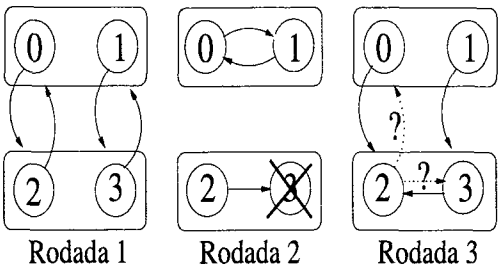

Figura 3.10: Prioridade no processo de tornar os testes isócronos.

maior prioridade com o qual está executando testes isócronos. Inicialmente, o valor da variável `iso` é igual a N para todo nodo.

Ao executar um teste, três situações distintas podem ocorrer: na primeira, o valor da variável `iso` do testador é memor que o valor da variável `iso` do nodo testado; na segunda, o valor da variável `iso` do testador é igual ao valor da variável `iso` do nodo testado; e na terceira, o valor da variável `iso` do testador é maior que o valor da variável `iso` do nodo testado.

Se a primeira situação ocorrer, significa que o testador tem prioridade sobre o nodo testado, então o nodo testado deve executar testes no cluster ao qual o nodo testador pertence e guardar na sua variável `iso`, o identificador que está na variável `iso` do nodo testador. Em outras palavras, o nodo testado executa testes isócronos com, no mínimo, o nodo testador e o nodo cujo identificador está guardado na variável `iso` do testador. A figura 3.11 ilustra esta situação. Nesta figura, em um primeiro momento, o nodo 3, possui em sua variável `iso` o identificador do nodo 0, significando que o nodo

3 e o nodo 0 executam testes isócronos. Considere o teste que o nodo 3 executa no nodo 2. Como o nodo 3 possui um identificador na variável `iso` menor que o identificador na variável `iso` do nodo 2, isto é, o nodo 3 tem maior prioridade, o nodo 2 deve responder ao teste do nodo 3, testar o cluster do nodo 3 na mesma rodada de testes e atualizar sua variável `iso` para 0, pois é o identificador que o nodo 3 possui em sua variável `iso`. Agora, os testes executados pelo nodo 2 se tornam isócronos com os testes do nodo 0 e do nodo 3.

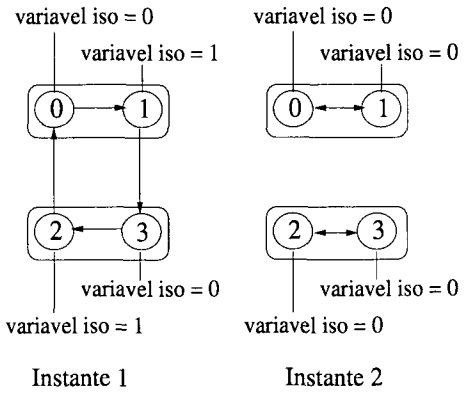

Figura 3.11: Comportamento dos nodos com maior prioridade.

A segunda situação é especial. Se ela ocorrer, dois casos podem ser desencadeados: primeiro, se o valor da variável `iso` de cada nodo é igual, porém diferente de N, o testador já possui testes isócronos com o nodo testado e continua os testes normalmente. No outro caso, se ambas as variáveis `iso`, do testador e do nodo testado, forem iguais a N, significa que os nodos estão

reinicializando o algoritmo. Neste caso, o nodo que terá maior prioridade será o de menor identificador. Se o nodo com maior prioridade for o testador, então a primeira situação se repete. Caso contrário, a terceira situação acontece. A figura 3.12 ilustra um sistema composto por 4 nodos executando o novo algoritmo. Nesta figura, na primeira rodada de testes, o nodo 0 e o nodo 1 estão falhos. Na segunda rodada de testes eles são reparados. Nesta rodada, num primeiro momento, ao valor da variável `iso` de ambos os nodos será atribuído o valor de N. Quando o nodo 0, por exemplo, executar um teste no nodo 1, ele terá maior prioridade, pois, as variáveis `iso` de ambos os nodos são iguais a N e o nodo 0 possui identificador menor que o nodo 1. Em um segundo momento, o nodo 1, então, deverá responder ao teste do nodo 0, executar um teste no cluster do nodo 0 e ambos os nodos deverão atualizar suas variáveis `iso` para 0, pois é o nodo 0 que tem maior prioridade.

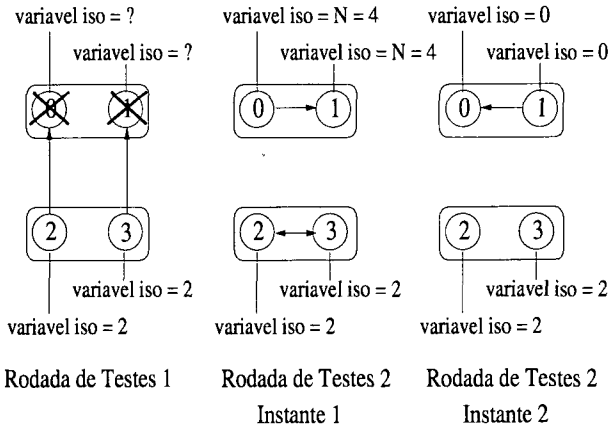

Figura 3.12: Comportamento dos nodos ao reinicializar o algoritmo.

Por fim, se a terceira situação ocorrer, isto é, o nodo testado possui maior prioridade, então ele recusa o teste, não respondendo ao mesmo, mas enviando uma mensagem para o nodo testador, informando em qual cluster o nodo testado está executando um teste e qual o valor de sua variável `iso`. Assim, na próxima rodada de testes o nodo testador poderá tornar seus testes isócronos com os testes do nodo testado, mudando a ordem do cluster a ser testado. Além disto, a variável `iso` do nodo testador recebe o valor da variável `iso` do nodo testado. A figura 3.13 mostra uma situação como esta. Nesta figura, na primeira rodada de testes, o nodo 3 testa o nodo 2, porém o nodo 2 tem maior prioridade. O nodo 2, então, manda uma mensagem para o nodo 3 informando que ele está testando o cluster de tamanho 2. Na próxima rodada de testes, isto é, na rodada 2, o nodo 3 testará o cluster de tamanho 1, ou seja, o nodo 2 novamente e os nodos estarão executando testes isócronos.

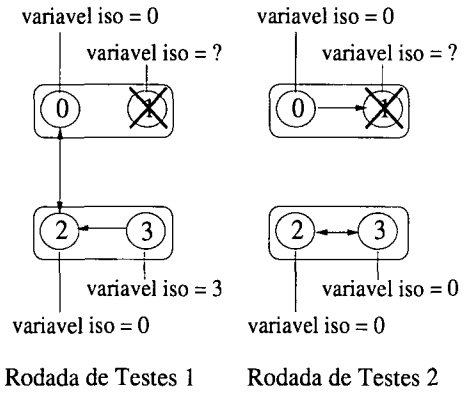

Figura 3.13: Comportamento dos nodos com menor prioridade.

Resumindo, para garantirmos uma estratégia de testes isócrona é preciso seguir os seguintes passos: primeiro, verifique o valor das variáveis `iso`, do nodo testador e do nodo testado, caso não se defina uma prioridade, isto é, se as variáveis `iso` possuírem o valor $= N$, terá prioridade aquele que tiver menor identificador. Caso o testador tenha maior prioridade, o nodo testado responde ao teste executado pelo testador, executa testes no cluster ao qual o nodo testador pertence e atualiza o valor da sua variável `iso`. Por outro lado, caso o nodo testado tenha maior prioridade, ele recusa o teste executado pelo testador e envia uma mensagem ao testador informando em qual cluster o nodo testado está executando testes e qual o valor da variável `iso` do nodo testado. Desta forma, na próxima rodada o testador executa testes no cluster de mesmo tamanho que o nodo testado e as variáveis `iso` do nodo testador e do nodo testado são iguais.

A seguir é definida, em pseudo-linguagem, uma função que especifica e executa o processo de tornar os testes isócronos. Considere o nodo i como o testador e o nodo j como o nodo testado.

```
FUNCAO ISO-TESTE (nodo i, nodo j)
{
    SE (i.iso < j.iso) /* nodo i tem maior prioridade */
    ENTAO
    {
        j.iso := i.iso;

        nodo i atualiza as informacoes de diagnostico sobre
        o cluster do nodo j;

        nodo j testa o cluster ao qual o nodo i pertence e
        atualiza as informações de diagnóstico sobre este cluster;
    }
    SENAO
        SE (i.iso = j.iso = N)
        ENTAO
            SE (i < j) /* nodos estao inicializando e nodo i tem
                          maior prioridade */
            ENTAO
            {
                i.iso := j.iso := i;

                nodo i atualiza informações de diagnóstico sobre
                o cluster do nodo j;

                nodo j testa o cluster ao qual o nodo i pertence e
                atualiza as informações de diagnóstico sobre este cluster;
            }
            SENAO  /* nodos estao inicializando e nodo j tem maior prioridade *
            {
                i.cluster := j.cluster+1;
                j.iso := j;
                i.iso := j.iso;
            }
        SENAO  /* nodo j tem maior prioridade */
        {
            i.cluster := j.cluster+1;
            i.iso := j.iso;
        }
}
```

O algoritmo Iso Hi-ADSD em pseudo-código segue abaixo:

```
Algoritmo Iso Hi-ADSD:
{para um nodo i}

REPITA
   PARA s:=1 ATE logN FACA
      REPITA
         j := proximo em c(i,s);
         SE  (j é sem-falha)
         ENTAO
         {
            ISO-TESTE(i,j); /* Funcao que tornar os testes
                               isócronos é chamada */
         }
      ATE (j ser sem-falha)
          OU (todos os nodos na c(i,s) testados falhos)
   FIM PARA;
PARA SEMPRE
```

3.3 Procedimento de Recuperação

Quando um nodo falho é reparado ele não possui as informações de diagnóstico corretas sobre o estado dos nodos no sistema, muitos eventos podem ter acontecido durante o período de tempo no qual o nodo estava falho. Além disto, o nodo que sofre a recuperação pode não voltar executando seus testes já isócronos com os dos outros nodos sem-falha, ele deve sofrer todo o processo de tornar os testes isócronos novamente.

Com o passar das rodadas de testes, o nodo que foi reparado, começa

a atualizar suas informações de diagnóstico, ou seja, este nodo irá possuir algumas informações corretas, pois já foram atualizadas e outras ainda incorretas, pois não foram atualizadas. A fim de evitar que este nodo, que sofreu a recuperação, passe as informações incorretas sobre o seu cluster a um nodo testador, é necessário especificar um procedimento de recuperação que permita indicar aos outros nodos quais as informações estão corretas e quais ainda não foram atualizadas.

Como especificado na estratégia de testes deste algoritmo, um nodo testado passa ao testador as informações de diagnóstico a respeito do cluster que o nodo testado pertence. Sendo assim, duas situações distintas podem ocorrer no momento em que o nodo que foi reparado é testado: ou o nodo que sofreu a recuperação consegue atualizar as informações de diagnóstico a respeito de seu cluster antes que seja testado, ou ainda não atualizou todas as informações a respeito de seu cluster.

Se a primeira situação ocorrer, o nodo que foi recuperado seguirá normalmente o algoritmo. Caso contrário, ele deverá avisar ao nodo testador que ainda não atualizou todas as informações a respeito de seu cluster e por isso só passará as informações corretas.

Para implementar este mecanismo foi construída uma tabela na qual o nodo reparado sabe quais as informações estão atualizadas e quais ainda permanecem desatualizadastas. Esta tabela é constítuida de dois campos: o

primeiro guarda o identificador de um nodo e o segundo marca se a informação de diagnóstico a respeito daquele nodo já foi atualizada ou não.

O segundo campo é implementado como um bit, setado para 1 se a informação de diagnóstico correspondente está atualizada e setado para 0 caso contrário. Chamamos este campo de `u-bit` (*updated bit*).

Quando um nodo inicializa o algoritmo todos os `u-bits` são setados para 0 e o valor da variável `iso`, ou seja, a variável que guarda com quem cada nodo possui testes isócronos, é inicializado com o valor de N. Com o passar das rodadas de testes, os nodos vão atualizando as informações a respeito dos nodos testados e o campo `u-bit` correspondente vai sendo setado para 1. Cada nodo testado deve verificar se o campo `u-bit` é igual a 1 para poder passar a informação do nodo correspondente.

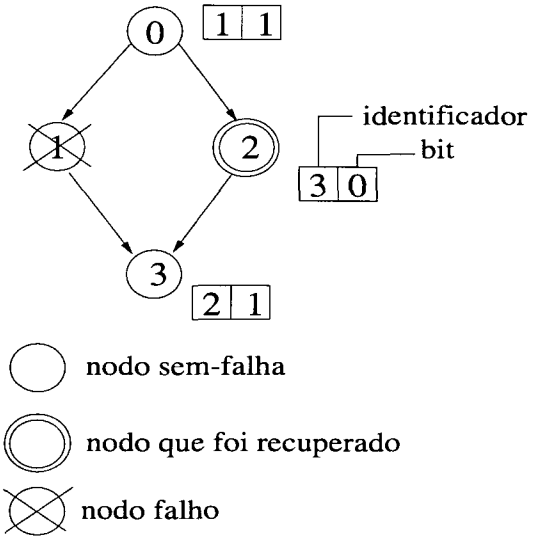

Figura 3.14: Procedimento de recuperação.

Considere o seguinte caso ilustrado na figura 3.14. Neste exemplo temos um sistema de 4 nodos. Note que o nodo 2 acaba de ser reparado e ainda não atualizou a informação a respeito do seu cluster, ou seja, a informação a respeito do nodo 3.

Quando o nodo 0 testa o nodo 2 e detecta que ele está sem-falha, o nodo 2 então, irá transferir as informações de diagnóstico a respeito do seu cluster, neste caso somente o nodo 3. Ao checar o `u-bit` correspondente, o nodo 2 irá avisar ao nodo 0 que a informação a respeito do nodo 3 ainda não está atualizada e por isso não irá transferir esta informação.

Para o nodo 2 atualizar o `u-bit` a respeito do nodo 3, basta ele testar o nodo 3 ou receber informação segura de diagnóstico de algum outro nodo a respeito do nodo 3. No momento que uma destas duas situações ocorrerem, o nodo 2 irá setar o `u-bit` correspondente ao nodo 3 para 1, e assim, poderá propagar a informação corretamente.

3.4 Provas Formais

Nesta seção são apresentadas provas formais que garantem o processo de tornar os testes isócronos, ou seja, se os nodos de um sistema inicializam o algoritmo executando testes em clusters de tamanhos diferentes a cada rodada, em um número finito de rodadas de testes todos os nodos estarão

executando testes em clusters de mesmo tamanho a cada rodada de testes. Além disso, é provado formalmente que o número máximo de rodadas de testes para que todos os nodos realizem o diagnóstico do sistema é de $2logN-1$. Todas as provas consideram que dada uma síndrome do sistema, nenhum evento pode ocorrer até que todo o processo de tornar os testes isócronos seja completado.

Seja a *árvore de disseminação de eventos* A_i, definida como a árvore na qual o nodo i é raiz e os nodos e folhas da árvore são os nodos sem-falha do sistema S. Existe uma aresta partindo do nodo j para o nodo k, se o nodo k otém informação de diagnóstico a respeito do nodo i a partir do nodo j. A figura 3.15 ilustra uma A_i genérica para um sistema composto por 8 nodos. A figura 3.16 ilustra uma A_i genérica para um sistema S qualquer. Em ambas as figuras, os nodos sem-falha dos sistemas fazem parte da árvore e estão referenciados como nodos pertencentes a alguma $c_{i,s}$, com s variando de 1 a $logN$.

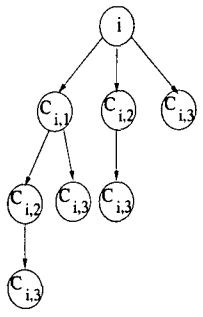

Figura 3.15: A_i genérica para um sistema composto por 8 nodos.

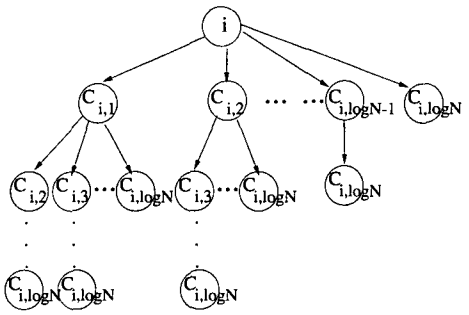

Figura 3.16: A_i genérica para um sistema S qualquer.

A lista de nodos $c_{i,s}$ é definida na seção de especificação do algoritmo, e tem como objetivo determinar para o testador (nodo i), qual nodo deve ser testado em cada cluster. Considere que um nodo j é chamado *filho* do nodo i se o nodo j é o primeiro nodo sem-falha testado pelo nodo i em uma dada $c_{i,s}$. Como, por definição, s pode ter valores de 1 até $logN$, então cada nodo i possui até $logN$ filhos. Por exemplo, em um sistema composto por 8 nodos, o nodo 0 possui três filhos: nodo 1 ($c_{0,1}$), nodo 2 ($c_{0,2}$) e nodo 4 ($c_{0,3}$).

Seja a *subárvore de disseminação de eventos* T_j, definida como uma subárvore da árvore A_i na qual o nodo j é raiz, os nodos e folhas da árvore são alguns dos nodos sem-falha do sistema S e o nodo i não pertence a subárvore T_j. Existe uma aresta partindo do nodo j para o nodo k, se o nodo k otém informação de diagnóstico a respeito do nodo i a partir do nodo j. A figura 3.17 ilustra três subárvores de A_i, a subárvore $c_{i,1}$, a subárvore $c_{i,2}$ e a subárvore $c_{i,3}$.

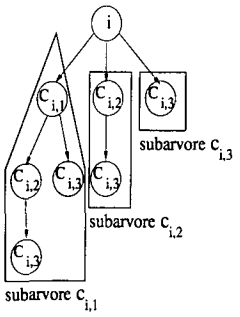

Figura 3.17: Um sistema composto por 8 nodos possui 3 subárvores T_j distintas.

A relação *descendente(nodo i, nodo j)* é definida $\forall$i,j $\in A_i$ como:

```
descendente(nodo i, nodo j) = Verdadeira
  se o nodo i testa o nodo j ou
  se existe algum nodo k tal que
    descendente(nodo i, nodo k) /\ descendente(nodo k, nodo j)
descendente(nodo i, nodo j) = Falsa
  caso contrário
```

Por exemplo, na figura 3.17 se descendente(nodo i, nodo $c_{i,1}$) e descendente(nodo $c_{i,1}$, nodo $c_{i,2}$) então descendente(nodo i, nodo $c_{i,2}$).

Teorema 1 *A estratégia de testes do algoritmo Iso Hi-ADSD garante que, em um número finito de rodadas de testes, todos os nodos sem-falha do sistema terão o mesmo valor armazenado nas respectivas variáveis* `iso`.

Prova:

No algoritmo Iso Hi-ADSD, por definição, cada nodo armazena na sua variável `iso` *um valor inteiro de 0 a* $N-1$*; apenas quando o algoritmo é inicializado a variável* `iso` *de cada nodo recebe o valor* N.

Seja o conjunto **FF** *definido como a união de todos os nodos sem-falha do sistema em uma determinada rodada de testes. Seja o conjunto* **SFF** *definido como um subconjunto de* **FF** *cujos nodos possuem o mesmo valor, diferente de N, nas respectivas variáveis locais* `iso`. *Considere ainda, um outro conjunto* **AFF** *que é definido como a diferença entre o conjunto* **FF** *e o conjunto* **SFF**, *ou seja,* **AFF={FF-SFF}**.

Inicialmente **SFF=∅** *e* **AFF={FF}**. *Devido a prioridade adotada ser a prioridade léxica, o nodo com maior prioridade ou é aquele que possui o menor valor em sua variável* `iso`, *ou se todos os nodos possuem* `iso`=*N, então o nodo com maior prioridade é o nodo com menor identificador. Seja o nodo com maior prioridade chamado de nodo i e sua variável* `iso` *chamada de* `iso-i`.

Considere agora que **SFF={nodo i}** *e* **AFF={FF-SFF}**, *ou seja,* **AFF={FF-nodo i}**. $\exists$ *subárvore de disseminação* $T_j \subset A_i$, *tal que* $\forall$*nodo* $x \in$ **FF**, *nodo* $x \in T_j$. *Sendo assim,* $\forall$*nodo* $x \in$ **FF**, *descendente (nodo i, nodo x) = Verdadeiro.*

Considere um nodo $x \in$ **FF** *e sua respectiva variável* `iso` *chamada de* `iso-x`. *Se o nodo x é testado pelo nodo i em uma determinada rodada, então* `iso-x` = `iso-i`, **SFF={SFF U nodo x}** *e* **AFF={FF-(SFF U nodo x}**. *Por outro lado, se o nodo x é testado por algum outro nodo* $\in$ **SFF**, *por exemplo nodo j, tal que descendente (nodo i, nodo j) = Verdadeiro, então a*

variável `iso-x` = `iso-j` = `i`, **SFF={SFF U nodo x}** *e* **AFF={FF-(SFF U nodo x)}**.

Como $\forall$*nodo* $x \in$ **FF**, *descendente (nodo i, nodo x) = Verdadeiro, todo nodo x ou é testado pelo nodo i, ou é testado por algum descendente do nodo* $i \in$ **SFF**. *Desta forma, em um número de rodadas de testes finita,* **SFF={FF}** *e* **AFF=**$\emptyset$.□

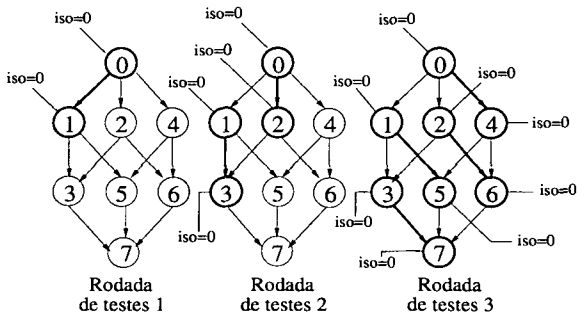

Figura 3.18: Procedimento de tornar os testes isócronos.

A figura 3.18 mostra um sistema composto por 8 nodos executando o algoritmo Iso Hi-ADSD. Na primeira rodada de testes o nodo 0, o qual possui o menor valor em sua variável `iso` e por isso tem maior prioridade, testa o nodo 1. O nodo 1 então, atualiza a variável `iso` para o valor de 0, seus testes se tornam isócronos com os testes do nodo 0 e o nodo 1 também passa a ter prioridade sobre os outros nodos. Na segunda rodada de testes, o nodo 0 testa o nodo 2, enquanto que o nodo 1 testa o nodo 3. Ambos, nodo 2 e nodo 3, atualizam suas respectivas variáveis `iso` para o valor de 0 e seus

testes se tornam isócronos com os testes do nodo 0 e nodo 1. Agora o nodo 2 e o nodo 3, também possuem prioridade sobre os outros nodos. Por fim, na terceira rodada de testes, os nodo 0, 1, 2 e 3 testam os nodos 4, 5, 6 e 7, respectivamente e todo o processo de tornar os testes isócronos se repete. Desta forma, neste exemplo, após $logN$ rodadas de testes todos os nodos estão testando clusters de mesmo tamanho a cada rodada de testes.

No exemplo anterior, o número total de rodadas de testes para que todos os nodos tivessem uma estratégia de testes isócrona foi de $logN$. Em outras situações, com alguns nodos falhos no sistema, o número total de rodadas de testes para que se complete o processo de tornar os testes isócronos pode ser maior.

Teorema 2 *Considere o menor valor possível da variável* `iso`. *Se dois nodos possuem este mesmo valor nas suas respectivas variáveis* `iso`, *então eles testam clusters de mesmo tamanho em cada rodada de testes.*

Prova

Considere que o nodo k é o nodo com maior prioridade no sistema, ou seja, o nodo k ou possui o menor valor em sua variável `iso` *ou se todos os nodos possuem suas respectivas variáveis* `iso` $= N$ *então o nodo k possui o menor identificador.*

Considere a árvore de disseminação A_k. *Todo nodo sem-falha do sistema*

$\in A_k$. $\forall$*nodo* $x \in A_k$ *descendente (nodo k , nodo x) = Verdadeiro. Considere dois outros nodos sem-falha, nodo i e nodo j. Como o nodo i e o nodo j* $\in$ A_k *então descendente(nodo k, nodo i) = Verdadeiro e descendente(nodo k, nodo j) = Verdadeiro.*

Quando o nodo k testa o nodo i, os testes do nodo i tornam-se isócronos com os testes do nodo k e `iso-i = iso-k`. *A partir da próxima rodada de testes o nodo i e o nodo k testam clusters de mesmo tamanho.*

Como descendente(nodo k, nodo j) = Verdadeiro então os testes do nodo j irão tornar-se isócronos com os testes do nodo k e em um número finito de rodadas de testes `iso-j = iso-k`. *Quando em uma rodada de testes a variável* `iso-j` *receber o valor da variável* `iso-k`, *na rodada de testes seguinte o nodo j e o nodo k irão testar clusters de mesmo tamanho. Desta forma, em um número finito de rodadas de testes* `iso-j = iso-i = iso-k` *e todos os nodos estarão testando clusters de mesmo tamanho a cada rodada de testes.*□

Lema 1 *Considere um novo evento que ocorre no nodo i. Os filhos do nodo i, em* A_i, *obtém informação de diagnóstico sobre o novo evento, no pior caso, nas próximas logN rodadas de testes.*

Prova

Qualquer nodo i, em um sistema S, possui até logN filhos. Cada filho do nodo i pertence a uma dada $c_{i,s}$, *com s variando de 1 a logN. Como pela*

estratégia de testes do algoritmo Iso Hi-ADSD, cada filho de um nodo testa seu pai em rodadas de testes diferentes porém consecutivas, então todos os filhos do nodo i irão obter informação de diagnóstico sobre o nodo i em, no máximo, logN rodadas de testes.□

Lema 2 *Se os nodos sem-falha de uma subárvore $c_{i,r}$ obtém informação sobre um novo evento no nodo i, em uma determinada rodada de testes, então todos os nodos sem-falha de $c_{i,r+1}$, pertencentes à mesma subárvore, obtém informação sobre o mesmo evento no nodo i na rodada de testes seguinte.*

Prova

No algoritmo Iso Hi-ADSD cada nodo sem-falha do sistema testa em rodadas de testes sucessivas um nodo de $c_{i,1}$, depois um nodo de $c_{i,2}$, ... depois um nodo de $c_{i,logN}$; em seguida um nodo de $c_{i,1}$ é novamente testado, e o processo se repete. Considere os nodos pertencentes a uma $c_{i,r}$. Estes nodos obtém informação sobre um novo evento no nodo i a partir de um nodo correspondente em $c_{i,1}$, $c_{i,2}$, ... $c_{i,r-1}$, além disso um dos nodos obtém a informação diretamente testando o nodo i.

Numa subárvore de disseminação qualquer, considere que a raiz, nodo j, é um filho do nodo i. Como o nodo j pertence a uma determinada $c_{i,r}$ e a estratégia de testes é isócrona, então podemos afirmar que na rodada de testes que o nodo j obtém a nova informação sobre o nodo i, todos os nodos

estão executando testes em suas respectivas $c_{nodo,r}$. *Na próxima rodada de testes todos os nodos estarão executando testes em* $c_{nodo,r+1}$, *ou seja, os nodos pertencentes a* $c_{i,r+1}$ *poderão obter a nova informação sobre o nodo i através de nodos pertencentes a* $c_{i,r}$, $c_{i,r-1}$ ..., $c_{i,1}$.

Em outras palavras, se o nodo j, que pertence a uma $c_{i,r}$, *obtém nova informação sobre o nodo i em uma determinada rodada de testes. Na rodada seguinte ele passa a informação para o nodo de* $c_{i,r+1}$. *Desta forma, todos os nodos de* $c_{i,r+1}$ *de uma subárvore otém a nova informação na rodada seguinte àquela em que todos os nodos de* $c_{i,r}$ *obtém.* □

A figura 3.19 ilustra um sistema composto por 8 nodos. Neste sistema considere que uma falha ocorre no nodo 0 em uma rodada de testes x. O nodo pertencente a $c_{0,1}$, nodo 1, testa e obtém informação de diagnóstico sobre o nodo 0. Seguindo a estratégia de testes isócrona, na rodada de testes $x+1$, os nodos pertencentes a $c_{0,2}$ testam os nodo de $c_{0,1}$ e obtém informação sobre o novo evento no nodo 0 através dos nodos $c_{0,1}$.

Lema 3 *Em até* $logN - 1$ *rodadas de testes um filho do nodo i em* A_i, *terá disseminado a informação sobre o evento no nodo i em toda a subárvore na qual é raiz.*

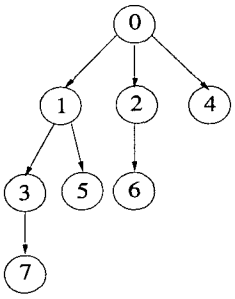

Figura 3.19: A cada rodada uma $c_{0,r}$ obtém informação sobre o nodo 0.

Prova

Os filhos do nodo i pertencem respectivamente a $c_{i,1}$, $c_{i,2}$, ..., $c_{i,logN}$ *e como mostrado anteriormente, são raízes de subárvores. Considere o nodo j, filho do nodo i, tal que nodo j* $\in c_{i,s}$. *O nodo j é raiz da subárvore de disseminação* $c_{i,s}$. *Se o nodo j descobre um novo evento no nodo i, ele precisa passar esta nova informação para o restante da sua subárvore, ou seja, nodos pertencentes a* $c_{i,s+1}$, $c_{i,s+2}$, ..., $c_{i,logN}$.

De acordo com o lema 2, se os nodos sem-falha de uma subárvore $c_{i,s}$ *obtém informação sobre um novo evento no nodo i, em uma determinada rodada de testes, então todos os nodos sem-falha de* $c_{i,s+1}$, *pertencentes à mesma subárvore, obtém informação sobre o mesmo evento no nodo i na rodada de testes seguinte.*

A subárvore de maior profundidade, é a subárvore $c_{i,1}$. *Quando o filho de i que pertence a* $c_{i,1}$ *descobrir um novo evento no nodo i, ele precisará passar esta nova informação para nodos pertencentes a* $c_{i,1+1}$, $c_{i,1+2}$, ..., $c_{i,logN}$.

Então, na primeira rodada de testes o nodo de $c_{i,1}$ passa a informação sobre o nodo i para o nodo de $c_{i,2} \in$ a subárvore $c_{i,1}$. Na próxima rodada de testes, ambos passam a informação sobre o nodo i para os nodos de $c_{i,3} \in$ a subárvore $c_{i,1}$, e assim por diante até que os nodos de $c_{i,logN} \in$ a subárvore $c_{i,1}$ descubram o evento no nodo i. Desta forma, no pior caso, em $logN - 1$ rodadas de testes todos os nodos pertencentes a subárvore $c_{i,1}$ terão diagnosticado o novo evento no nodo i.□

Teorema 3 *A latência do algoritmo Iso Hi-ADSD é de no máximo $2logN-1$ rodadas de testes.*

Prova:

De acordo com o lema 1, em no máximo $logN$ rodadas de testes os filhos do nodo i terão diagnosticado um novo evento no nodo i. Conforme o lema 3 em até $logN-1$ rodadas de testes um filho do nodo i em A_i, terá disseminado a informação sobre o evento no nodo i em toda a subárvore na qual é raiz.

Desta maneira, no pior caso, se o filho do nodo i que leva $logN-1$ rodadas de testes para disseminar uma nova informação em toda a sua subárvore levar $logN$ rodadas de testes para descobrir o novo evento no nodo i, então teremos $logN + logN - 1 = 2logN - 1$ rodadas de testes para que todos os nodos do sistema S realizem o diagnóstico do novo evento no nodo i.□

Capítulo 4

Resultados Experimentais

Neste capítulo são apresentados resultados experimentais de diagnóstico usando o algoritmo Iso Hi-ADSD. Estes resultados já foram apresentados em [24] e foram obtidos através de simulações em redes de tamanhos variados. As simulações foram executadas usando a linguagem de simulação de eventos discretos SMPL [25]. Os nodos foram modelados como *facilities* do SMPL. Três tipos de eventos foram definidos: teste, falha e reparo.

Resultados de três tipos de experimentos serão apresentados. No primeiro experimento é simulada uma situação em que inicialmente os nodos sem-falha de um sistema executam teste de maneira não isócrona, ou seja, os nodos testam clusters de tamanhos diferentes, determinados aleatóriamente, em uma mesma rodada de testes. Os nodos deste sistema executam o algoritmo Iso Hi-ADSD. Ao final deste experimento é comprovado que os testes dos nodos

tornam-se isócronos. Resultados deste experimento mostram que, mesmo se novos eventos ocorram durante a execução, o novo algoritmo consegue manter a estratégia de testes do sistema isócrona.

No segundo experimento é simulada uma situação que leva ao pior caso a latência do algoritmo Iso Hi-ADSD, ou seja, os nodos sem-falha do sistema já executando testes de maneira isócrona demoram o maior tempo possível para conseguir completar o diagnóstico. O terceiro experimento ilustra o número máximo de testes, em uma única rodada, executados por nodos rodando o algoritmo Iso Hi-ADSD.

4.1 Tornando a Estratégia de Testes Isócrona

O objetivo deste experimento é ilustrar como os nodos sem-falha de um sistema, executando o algoritmo Iso Hi-ADSD, comportam-se até conseguirem tornar à execução dos testes isócrona.

Este experimento é dividido em duas partes: na primeira parte, em uma rede de 8 nodos, considere que os nodos sem-falha estão inicialmente executando testes de forma não isócrona. É ilustrado então, detalhadamente, o comportamento dos nodos sem-falha durante o processo de tornar os testes isócronos.

Na segunda parte, em uma rede de 64 nodos, considere que os nodos

sem-falha estão inicialmente executando testes de forma não isócrona. É mostrado então, o número total de nodos que passam a executar testes de forma isócrona a cada rodada de testes. Ainda neste experimento, após os nodos terem tornado a execução de seus testes isócrona, ocorrem dois eventos no sistema. É mostrado então, o comportamento dos nodos a partir do evento até o processo de tornar os testes isócronos se completar.

Considere uma rede de 8 nodos executando testes de maneira não isócrona. A tabela 4.1 ilustra o comportamento dos nodos durante o processo de tornar os testes isócronos. Nesta tabela, a linha 1 mostra o cluster em que cada nodo começa a execução dos testes. A linha 2 mostra a primeira rodada de testes após o começo da execução do algoritmo. Note que o nodo 1 muda seus testes do cluster de tamanho dois para o cluster de tamanho um. Esta mudança ocorre porque o nodo 0 testa o nodo 1 e o nodo 0 possui maior prioridade, neste caso o nodo 1 deve testar o cluster ao qual o nodo 0 pertence na mesma rodada de testes e atualizar a variável `iso` para 0. O mesmo ocorre com o nodo 6, o qual é testado pelo nodo 2 e muda seus testes do cluster de tamanho dois para o cluster de tamanho quatro.

A terceira linha mostra a segunda rodada de testes após o começo da execução do algoritmo. Nesta linha o nodo 2 muda seus testes do cluster de tamanho um para o cluster de tamanho quatro, pois o nodo 0 testa o nodo 2 e o nodo 0 possui maior prioridade. Ainda na segunda rodada de testes, o

Rodada / Tamanho do cluster								
Nodos	0	1	2	3	4	5	6	7
Rodada de testes 0	1	2	4	1	2	4	2	1
Rodada de testes 1	1	1	4	1	2	4	4	1
Rodada de testes 2	2	2	2	2	1	2	1	2
Rodada de testes 3	4	4	4	4	4	4	4	4

Tabela 4.1: Comportamento dos nodos durante o processo de tornar os testes isócronos.

nodo 5 testa o nodo 1. Como o nodo 1 possui maior prioridade sobre o nodo 5, o nodo 1 recusa o teste e manda uma mensagem para o nodo 5 informando em qual cluster está executando testes nesta rodada e qual o valor de sua variável `iso`. Assim na próxima rodada de testes, o nodo 5 deve incrementar de uma unidade o valor recebido na mensagem e então os testes do nodo 5 se tornarão isócronos com os testes do nodo 1. Por fim, na última linha, todos os nodos estão executando testes em cluster de mesmo tamanho, ou seja, todos os nodos estão executando testes de maneira isócrona.

Considere agora, uma rede 64 nodos rodando o novo algoritmo. Inicialmente os nodos estão executando testes de maneira não isócrona e todos os nodos estão sem-falha. A tabela 4.2 mostra o número total de nodos que tornam a execução de seus testes isócrona a cada rodada de testes. Na primeira

Rodadas de Testes	*# de Nodos com Testes Isócronos*
0	0
1	13
2	34
3	57
4	64

Tabela 4.2: Um número cada vez maior de nodos executam testes isócronos.

rodada de testes, 13 nodos estão executando os testes de maneira isócrona. Na segunda rodada de testes, 34 nodos estão executando os testes de maneira isócrona. E assim por diante, até a quarta rodada de testes, na qual todos os 64 nodos do sistema estão executando testes de maneira isócrona.

O gráfico na figura 4.1 ilustra o mesmo experimento que é mostrado na tabela 4.2, porém, após o processo de tornar os testes isócronos, dois eventos ocorrem: ambos os nodos 0 e 32 falham. Apesar destes novos eventos, o sistema mantém a estratégia de testes isócrona. Suponha que os nodos 0 e 32 voltem a ficar sem-falha na rodada de testes x. Como as variáveis `iso` de cada um serão incializadas com o valor de N, indicando que eles possuem a menor prioridade possível, na rodada de testes $x + 1$ quando os nodos 0 e 32 forem testados, independente do nodo testador, eles tornarão seus testes isócronos com os testes do testador.

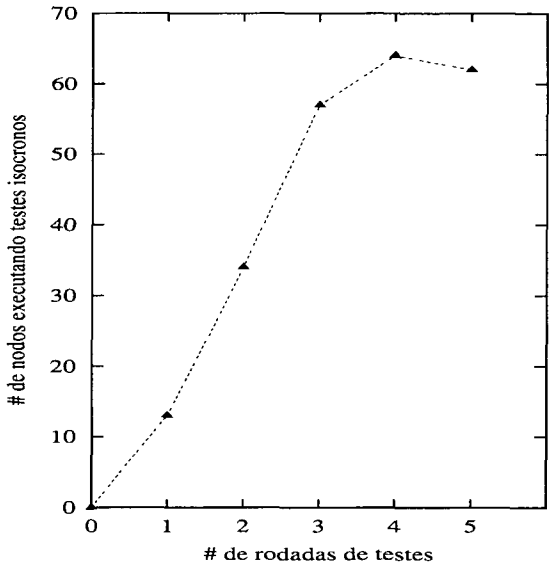

Figura 4.1: Número de nodos executando testes isócronos.

4.2 Latência Máxima do Algoritmo Iso Hi-ADSD

O objetivo deste experimento é ilustrar uma situação que leva ao pior caso da latência no novo algoritmo, ou seja, queremos mostrar neste experimento uma situação em que após os nodos terem tornado seus testes isócronos, eles demorem o maior tempo possível para completar um diagnóstico sobre o sistema.

Considere um sistema com 64 nodos. Para simular o pior caso da latência, um evento deve ocorrer no nodo x um instante de tempo depois que este nodo foi testado por um nodo y. Neste experimento, com o propósito de alcançarmos a latência máxima, ambos os nodos x e y devem pertencer ao

mesmo cluster com tamanho dois.

A figura 4.2 ilustra um sistema com 8 nodos executando o novo algoritmo, considerando que a estratégia de testes já está isócrona. No primeiro intervalo de testes todos os nodos estão testando clusters de tamanho um. Por exemplo, o nodo 6 está testando o nodo 7, o qual pertence ao cluster de tamanho um em relação ao nodo 6. Logo após o nodo 6 ter testado o nodo 7 um evento ocorre no nodo 7. Então, levará $logN = 3$ rodadas de testes para o nodo 6 realizar o diagnóstico do novo estado do nodo 7. Em exatamente mais uma $(logN + 1 = 4)$ rodada de testes, o nodo 4 irá diagnosticar o novo estado do nodo 7 a partir do nodo 6. Por fim, em mais uma $(logN + 1 + 1 = 2logN - 1)$ rodada de testes o nodo 0 irá diagnosticar o novo estado do nodo 7 a partir do nodo 4.

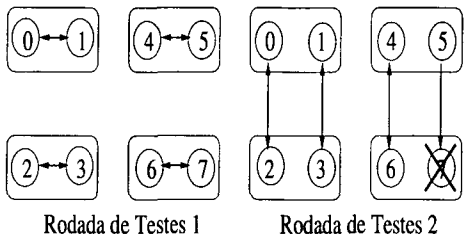

Figura 4.2: Uma situação que leva à latência máxima do novo algoritmo.

A tabela 4.3 mostra o número de rodadas de testes necessárias para que todos os nodos no sistema realizem o diagnóstico de um evento, o qual corresponde a falha do nodo 63. Neste experimento o nodo 63 muda de estado

logo após o nodo 62 ter realizado um teste no nodo 63. Após uma rodada de testes, somento um nodo sem-falha diagnosticou o evento. Após três rodadas de testes, 7 nodos diagnosticaram o evento. Após $logN - 1 = 5$ rodadas de testes, 31 nodos diagnosticaram o evento e por fim, após $2logN - 1 = 11$ rodadas de testes, todos os nodos sem-falha completaram o diagnóstico do evento.

# de rodadas após o Evento	*# de Nodos que Diagnosticaram o Evento*
1	1 nodo
3	7 nodos
5	31 nodos
7	33 nodos
9	39 nodos
11	63 nodos

Tabela 4.3: Nodos que completaram o diagnóstico por rodada de testes.

A tabela 4.4 mostra uma comparação entre a latência máxima, dada em rodadas de testes, do novo algoritmo e a latência máxima dos outros algoritmos estudados. Este experimento foi conduzido em redes de diversos tamanhos, começando em uma rede de 8 nodos, indo para uma rede de 16 nodos, e assim por diante, até alcançarmos uma rede de 512 nodos.

A latência máxima do algoritmo Iso Hi-ADSD é um resultado com impacto prático significativo, pois se considerarmos que uma rodada de testes terá 30 segundos, o tempo total que o novo algoritmo vai precisar para diagnosticar um evento em uma rede de 512 nodos, por exemplo, baixa de

Latência(#Rodadas)	*8*	*16*	*32*	*64*	*128*	*256*	*512*
ADSD	8	16	32	64	128	256	512
Hi-ADSD	9	16	25	36	49	64	81
Hi-ADSD with Detours	9	16	25	36	49	64	81
Iso Hi-ADSD	5	7	9	11	13	15	17

Tabela 4.4: Comparação da latência máxima nos algoritmos estudados.

256 minutos (ADSD), 40 minutos e 30 segundos (Hi-ADSD e Hi-ADSD with Detours) para apenas 8 minutos e 30 segundos (Iso Hi-ADSD). Além disto, os resultados da tabela 4.4 mostram que quando o tamanho de uma rede dobra, a latência máxima do algoritmo Iso Hi-ADSD é aumentada apenas de 2 unidades.

4.3 Número Máximo de Testes Necessários para o algoritmo Iso Hi-ADSD

O objetivo deste experimento é ilustrar o número máximo de testes necessários para que todos os nodos sem-falha, executando o novo algoritmo, completem um diagnóstico.

O número máximo de testes do novo algoritmo ocorre quando $N/2$ nodos estão falhos e pertencem ao mesmo cluster. Além disto, os outros $N/2$ nodos sem-falha devem estar testando aquele cluster na mesma rodada de testes. A figura 4.3 ilustra esta situação.

Na simulação realizada foi considerado um sistema como o ilustrado na figura 4.3, ou seja, o nodo 4, nodo 5, nodo 6 e nodo 7 estão falhos e pertencem

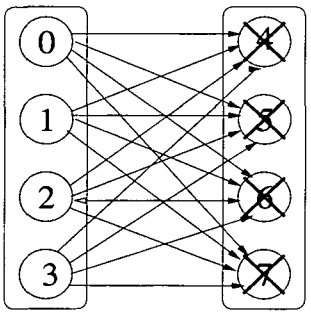

Figura 4.3: Pior caso do número de testes no algoritmo Iso Hi-ADSD.

ao mesmo cluster, enquanto que os $N/2$ nodos restantes; nodo 0, nodo 1, nodo 2 e nodo 3 estão sem-falha e executam testes naquele cluster na mesma rodada de testes.

Podemos notar, através deste experimento, que na mesma rodada de testes todos os nodos sem-falha (nodo 0, nodo 1, nodo 2 e nodo 3) estão testando um cluster de tamanho $N/2$ e todos os nodos que pertencem a este cluster estão falhos ($N/2$ nodos falhos). Temos então, 4 nodos sem-falha executando 4 testes cada um, o que resulta em um total de 16 testes em uma única rodada de testes, ou seja, um total de $N^2/4$ testes em uma única rodada de testes.

4.4 Considerações sobre a Implementação Prática do Algoritmo Iso Hi-ADSD

Nesta seção, são analizadas algumas considerações para a melhor implementação prática do algoritmo Iso Hi-ADSD.

4.4.1 Determinação do Intervalo e da Rodada de Testes

Considere uma rede Ethernet que transmite dados a 100Mbps e que as máquinas desta rede se comunicam usando o protocolo TCP/IP. Experimentos realizados mostram que ao utilizarmos o comando *ICMP echo request/reply* as máquinas testadas demoravam em média cerca de 2 milisegundos para responder.

Considere que um nodo sem-falha demora 10 milisegundos para detectar se um outro nodo está falho ou sem-falha. Como no pior caso em uma rodada de testes, um nodo sem-falha testa $N/2$ nodos então o tamanho máximo necessário para uma rodada de testes será:

$$N/2 \; testes \; * \; 10 \; milisegundos \; = \; 5N \; milisegundos$$

Se $N = 100$, então o tamanho máximo de uma rodada de testes será de 500 milisegundos, ou seja, 0.5 segundos. Para facilitar os cálculos vamos considerar que cada rodada de testes seja de 1 segundo.

4.4.2 Análise do Impacto do Algoritmo no Desempenho da Rede

Considere uma rede Ethernet, com 100 máquinas, que transmite dados a 100Mbps. Considere também, rodadas de teste de 1 segundo. Como no pior caso o número total de testes do algoritmo Iso Hi-ADSD é de $N^2/4$ testes

em uma única rodada de testes e cada mensagem trocada entre os nodos em um teste é de 100 bytes, então temos:

Pior caso
Usados:

$$100^2 = 10000\ testes * 100\ bytes\ transmitidos * 1$$

$$= 1000000 = 10^6\ bytes\ usados\ em\ 1\ segundo$$

Temos disponíveis:

$$100000000\ bytes * 1\ segundo =$$

$$= 10^8\ bytes\ por\ segundo$$

Usado/disponível:

Usamos 10^{-2} bytes por segundo da capacidade total de transmissão da rede no pior caso.

Melhor Caso

Todos os nodos da rede estão sem-falha, ou seja, 100 nodos sem-falha. Eles estão executando testes isócronos. Como todos estão sem-falha, cada nodo realiza um teste e recebe as informações necessárias.

Usados:

$$100\ mensagens\ =\ 10000\ bytes\ =\ 10^4\ bytes\ por\ segundo$$

Disponíveis: 10^8 bytes por segundo

Usados/disponível:

Usamos 10^{-4} bytes por segundo da capacidade de transmissão de dados da rede no melhor caso.

Como numa rede Ethernet há a possibilidade de colisões de mensagens ocorrerem então, se logo após o início de uma rodada, todas as máquinas tentarem testar simultaneamente, as mensagens irão colidir. Para evitar este problema basta introduzir um pequeno atraso (até 100 milisegundos) aleatório para diminuir a probabilidade de colisões.

Capítulo 5

Conclusão

Algoritmos de diagnóstico em nível de sistemas têm sido usados para a construção de sistemas confiáveis em diferentes áreas de aplicação.

Este trabalho apresentou um novo algoritmo de diagnóstico hirárquico adaptativo e distribuído em nível de sistema. O algoritmo Iso Hi-ADSD permite à todos nodos sem-falha no sistema completar um diagnóstico em, no máximo, $2logN - 1$ rodadas de testes. Algoritmos anteriores tais como, Hi-ADSD e Hi-ADSD with Detours, apresentam latência de log^2N rodadas de testes, no pior caso.

Para melhorar a latência no pior caso, ou seja, para diminuir o tempo máximo no qual os nodos sem-falha completam o diagnóstico do sistema, o algoritmo Iso Hi-ADSD emprega uma estratégia de testes chamada de *estratégia de testes isócrona*, a qual força todos os nodos sem- falha do sistema

a executarem testes em clusters de mesmo tamanho a cada rodada de testes.

Para manter a estratégia de testes isócrona, o algoritmo emprega dois princípios básicos: um nodo testado deve testar o cluster ao qual o nodo testador pertence na mesma rodada de testes; um nodo somente aceita testes de acordo com um mecanismo de prioridade léxica. Assume-se que cada intervalo de testes é grande o suficiente para um nodo sem-falha testar e determinar o estado de até $N/2$ nodos. Todos os nodos sem-falha conhecem o início de cada intervalo de testes.

Foram apresentadas provas formais que comprovam o processo de tornar a execução dos testes isócrona, ou seja, se os nodos de um sistema inicializam o algoritmo executando testes em clusters de tamanhos diferentes a cada rodada, em um número finito de rodadas de testes todos os nodos estarão executando testes em clusters de mesmo tamanho a cada rodada de testes. É também provado que o número máximo de rodadas de testes para que todos os nodos realizem o diagnóstico do sistema é de $2logN - 1$.

Além dos teoremas matemáticos, foram apresentados resultados experimentais obtidos usando-se a linguagem de eventos discretos SMPL. Estes resultados confirmam que ao executarem o algoritmo Iso Hi-ADSD todos os nodos sem-falha do sistema tornarão seus testes isócronos, além disto foram simulados o pior caso da latência e o número máximo de testes do algoritmo Iso Hi-ADSD.

Trabalhos futuros incluem estudos para diminuir o pior caso do número de testes e diminuir o tempo de cada rodada de testes, o impacto de falhas dinâmicas no algoritmo e uma implementação Java/SNMP que permita o uso do algoritmo Iso Hi-ADSD na gerência de falhas de redes locais.

Bibliografia

[1] P. Jalote, *Fault Tolerance in Distributed Systems.* Englewood Cliffs, N.J.: Prentice Hall, 1994.

[2] S. Mullender, *Distributed Systems.* ACM Press Frontier Series, 1993.

[3] G. Masson, D. Blough, & G. Sullivan, "System Diagnosis," *Fault-Tolerant Computer System Design,* ed. D.K. Pradhan, Prentice-Hall, 1996.

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

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

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

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

[8] R.P. Bianchini, & R. Buskens, "An Adaptive Distributed System-Level Diagnosis Theory," *Proc. FTCS-21*, pp. 222-229, 1991.

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

[10] E.P. Duarte Jr., & T. Nanya, "A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm," *IEEE Transactions on Computers*, Vol. 47, pp. 34-45, No. 1, Jan 1998.

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

[12] M. Malek, "A Comparison Connection Assigment for Diagnosis of Multiprocessor Systems," *Proc. 7th Int'l Symp. Computer Architecture*, pp 31-36, 1980.

[13] K.Y. Chwa, & 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.

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

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

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

[17] S. Lee & K. G. Shin, "Probabilistic Diagnosis of Multiprocessor Systems," *ACM Computing Surveys*, Vol.26, No.1, March 1994.

[18] E.P. Duarte Jr., & T. Nanya, "Multi-Cluster Adaptive Distributed System-Level Diagnosis Algorithms," *IEICE Technical Report FTS 95-73*, 1995.

[19] M. Malek, & J. Maeng, "Partitioning of Large Multicomputer Systems for Efficient Fault Diagnosis," *Proc. FTCS-12*, pp. 341-348, 1982.

[20] A. Bagchi, "A Distributed Algorithm for System-Level Diagnosis in Hypercubes," *Proc. 1992 IEEE Workshop Fault-Tolerant Parallel and Distributed Systems*, pp. 106-113, 1992.

[21] M. Malek, & J. Maeng, "Partitioning for Efficient Consensus," *Proc. 26th Hawaii Int'l Conf. System Sciences*, Vol. 2, pp. 438-446, 1993.

[22] J. Altman, F.Balbach, & A. Hein, "An Approach for Hierarchical System-Level Diagnosis of Massively Parallel Computers Combined with a Simulation-Based Method for Dependability Analysis," *Proc. Firts European Dependable Computing Conf., Lecture Notes in Computer Science*, Vol. 852, pp. 371-385, 1994.

[23] E.P. Duarte Jr., A. Brawerman, & L.C.P. Albini, "An Algorithm for Distributed Hierarchical Diagnosis of Dynamic Fault and Repair Events," *Proc. 7th. International Conference on Parallel and Distributed Systems*, 2000.

[24] A. Brawerman, & E.P. Duarte Jr., "A Synchronous Testing Strategy for Hierarchical Adaptive Distributed System-Level Diagnosis," *Proc. 1st. IEEE Latin-American Test Workshop*, pp. 154-161, 2000.

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

Apêndice A

Programa de Simulação

Segue em anexo o código em linguagem C do algoritmo Iso Hi-ADSD.

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

/* events:  */

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

/* struct of a node:  */

typedef struct {
   int id,      /* SMPL facility identifier */
       *tst,    /* TESTED-UP array */
       cont,    /* number of tests */
       change,  /* number of events detected */
       t_round,
       cont_tests,
       cluster, /* cluster to test */
       sync,    /* synchronized with who */
       tested;  /* last node which was tested */
```

```
  float time_tst; /* time in which a node was tested */
} tnode;

FILE *out;   /* output file */
FILE *graf; /* graphic's input */
FILE *cont;
tnode *node;

static int N;  /* Number of Nodes passed likes parameter */
static char nomearq[20];
static char name[20];
static int sabem=0, new_event=0,
         prev_round=0; /* variables to save the graphics file */

void make_faulty(int a)
{
  int i,cont=0,aux,b;
  int nos[64];

  for (b=0;b<64;b++)
    {
      nos[b]=0;
    }

  for (i=a;;i++)
    {
      srand(i);
      aux = (int)(64.0*rand()/(RAND_MAX+1.0));
      if (nos[aux]!=1)
  {
    nos[aux]=1;
    schedule(fault, 10.0,aux);
    cont++;
  }
```

```
      if (cont==32)
break;
    }
  printf("\n rounds needed for %d faulty nodes => %d\n",i,cont);
}

/* 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;
  }

void record2(token,next,time,round)
int token,
    next;
real time;
int round;
{
 if (new_event == 1)
 { new_event = 0;
   if (prev_round != 0)
     fprintf(graf, "%d %d\n", prev_round,know);
```

```
   fprintf(graf, "%d %d\n", round-1,0);
   know = 1;
   prev_round = round;
 }
 else
{  know++;
   if (round != prev_round)
   { fprintf(graf, "%d %d\n", prev_round,know-1);
     prev_round = round;
   }
}
}

void record(token,next,time,round)
int token,
    next;
real time;
int round;
{
 fprintf(out, "node %d got new info on node %d, at time %4.1f, round %d\n",
      token, next, time, round );
}


unsigned int find_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);
}

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 */


void getinfo(int token, int N, int next_to_test, int j, int flag)
{
  static int i,
    next_to_copy,
    new_j;

    printf("node %d tests %d in cluster %d at time %5.1f, j=%d\n",
            token, next_to_test,node[token].cluster,time(),j);

  /* INCREASE */
  node[token].cont_tests++;

  while (status(node[next_to_test].id) == 1)  /* faulty */
```

```
  {
    /* "verify if it is a new info -> change++" */
    if (node[token].tst[next_to_test] == 0) {
      node[token].tst[next_to_test] = 1;
      node[token].change++;
      record(token, next_to_test, time(), node[token].cont);
      record2(token, next_to_test, time(), node[token].cont); }


    /* increase(j,cluster);  */
    j++;
    if (j == elev(node[token].cluster-1))
      {
j=0;
node[token].cluster++;
/*if (node[token].cluster == logtwo(N)+1)
  node[token].cluster = 1; */
break;
      }

    next_to_test = c_is (token, node[token].cluster,j);
    printf("node %d tests %d in cluster %d at time %5.1f, j=%d\n",
            token, next_to_test,node[token].cluster,time(),j);

    /* INCREASE */
    node[token].cont_tests++;
    /* Last node to be tested */
    node[token].tested=next_to_test;

  }  /* exits loop only when a fault-free node is found */

  /* verify if info about fault-free tested node is new */
  if (status(node[next_to_test].id) == 0)
  {
```

```
    if (node[token].tst[next_to_test] == 1)
      {
node[token].tst[next_to_test] = 0;
node[token].change++;
record(token, next_to_test, time(), node[token].cont);
record2(token, next_to_test, time(), node[token].cont); }


    /* verify if it has new info then copy cluster's information */

    new_j = j+1;
    if (new_j == elev(node[token].cluster-1))
      new_j = 0;

    while (new_j != 0) /* node has tested from 0 to j */
      {
next_to_copy = c_is(token, node[token].cluster, new_j);
if (node[token].tst[next_to_copy] !=
    node[next_to_test].tst[next_to_copy])
  {
    node[token].change++;
    record(token, next_to_copy, time(), node[token].cont);
    record2(token, next_to_copy, time(), node[token].cont);
    node[token].tst[next_to_copy]=node[next_to_test].tst[next_to_copy];
  }
new_j++;
if (new_j == elev(node[token].cluster-1))
  new_j = 0;
      }

    /* If nobody tested that node*/
    if (flag==0)
    {
      node[token].cluster++;
```

```
      if((token>next_to_test) && (node[token].sync<node[next_to_test].sync))
  {
    node[next_to_test].sync=node[token].sync;
    node[next_to_test].cluster=node[token].cluster;
   }
/* if(token<next_to_test)
   node[token].cluster=node[next_to_test].cluster+1;*/
/*else*/

    }
  }
}

/* main loop */

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

{
static int event,          /* event identifier, natural number */
    i,j=0,
    token,          /* node identifier, natural number */
    next_to_test,
    next_to_copy,
    new_j,
    r,
    ind, auxi,      /* auxiliar variables to make itoa */
    flag, token_aux;  /* it's used to know the test_me_now part */

static char fa_name[5];

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

```
sprintf(nomearq,"%s","graf");
   if ((graf= fopen(nomearq,"w"))  == NULL) {
         fprintf(out, "ERROR - Results' file was not openned");
         fflush(graf);
         exit(-1);
   }

sprintf(nomearq,"%s","relat");
if ((out = fopen(nomearq,"w"))  == NULL) {
      fprintf(out, "ERROR - Results' file was not openned");
      fflush(out);
      exit(-1);
}

sprintf(nomearq,"%s","contagem");
if ((cont = fopen(nomearq,"w"))  == NULL) {
      fprintf(cont, "ERROR - Results' file was not openned");
      fflush(cont);
      exit(-1);
}

smpl(0,"Hi-ADSD Simulation");
/* trace(2); */
reset();
stream(1);

/* initialization */


    node = (tnode *) malloc(sizeof(tnode)*N);

    for (i=0; i<N; i++) {
      node[i].cluster = 1; /*+(int)(9.0*rand()/(RAND_MAX+1.0));*/
```

```
        printf("node= %d cluster= %d\n",i,node[i].cluster);
        node[i].cont    = 0;
        node[i].change  = 0;
        node[i].tst     = (int *) malloc (sizeof(int) * N);
        node[i].cont_tests=0;
        node[i].iso=N;
        node[i].tested=N;
        node[i].time_tst=30.0;

      memset (fa_name, '\0', 5);

      ind = 0;
      auxi = i;

      if (auxi > 99)
       { fa_name[ind] = auxi / 100 + '0';
         auxi = auxi % 100;
         ind++; }

      if (auxi > 9)
       { fa_name[ind] = auxi / 10 + '0';
         auxi = auxi % 10;
         ind++; }

      fa_name[ind] = i + '0';

      node[i].id      = facility(fa_name,1);

      for (j=0; j<N; j++)
         node[i].tst[j]=0;
    }

/* schedule initialization */
```

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


/* make_faulty(atoi(argv[3]));*/
 schedule(fault, 0.0, 3);
 schedule(fault, 0.0, 1);
 schedule(fault, 0.0, 2);
 schedule(fault, 0.0, 5);
 /*schedule(fault, 0.0, 6); */
 while(time() < 500.0) {

  cause (&event, &token);
/*  token = token % N;  */

  switch(event) {

  case test:
if (status(node[token].id) != 0) break;

/*if (node[token].cluster == 2)
          {
            fprintf(cont,"COUNTER %d - %4.1f = %d\n", token, time(),
                    node[token].cont_tests);
           node[token].cont_tests=0;
    }*/

j=0;
node[token].tested =  c_is (token, node[token].cluster,j);
node[token].time_tst=time();
flag=0; token_aux=0;

/* Priority tests */
while ((token_aux<N) && (flag!=1))
```

```
{
  if (token==token_aux)
    token_aux++;
  if ((token==node[token_aux].tested) && (time()==node[token_aux].time_tst))
    {
      if (node[token_aux].sync<node[token].sync)
{
  flag=1;
  node[token].sync=node[token_aux].sync;
  node[token].cluster=find_cluster(token_aux,token);
  node[token].tested=c_is(token, node[token].cluster,j);
  getinfo(token,N,node[token].tested,j,flag);
}
    }
  token_aux++;
}
/* End of tests */

if (flag==0)
  getinfo(token,N,node[token].tested,j,flag);
else
  /* increase cluster to next test  */
  /* j always begins in 0 */
  node[token].cluster++;

if (node[token].cluster == logtwo(N)+1)
    {
      node[token].cluster = 1;
      fprintf(cont,"COUNTER %d - %4.1f = %d\n", token, time(),
              node[token].cont_tests);
      node[token].cont_tests=0;
    }
/* update tests' counter and prepare next test */
node[token].cont++;
```

```
schedule(test, 30.0, token);
break;

   case fault:
        new_event = 1; /* to save in graphics file */
fprintf(out, "node %d becomes faulty at time %5.1f\n", token, time());
        r = request(node[token].id,token,0);
        if (r != 0) {
  printf("Not possible to force fault");
          fflush(stdout);
  exit(0);
        }
        else /*schedule(repair, 210.0, token); schedule(repair, 210.0, 0);*/
        break;

    case repair:
        new_event = 1; /* to save in graphics file */
fprintf(out, "node %d is repaired at time %5.1f\n", token, time());
release(node[token].id, token);
        schedule(test, 0.0, token);    /* i is token number */
        break;
  }
 }
  record2(1, 2, time(), 5000); /* force the last line of file graf */
}
```