ROVERLI PEREIRA ZIWICH

DETECÇÃO DISTRIBUÍDA DE ALTERAÇÕES EM SISTEMAS COM CONTEÚDO REPLICADO UTILIZANDO DIAGNÓSTICO BASEADO EM COMPARAÇÕES

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Elias Procópio Duarte Jr.

CURITIBA
2004

Agradecimentos

Eu não poderia terminar este trabalho sem agradecer a todos que me ajudaram e me apoiaram durante todo meu mestrado. Em primeiro lugar, gostaria de agradecer ao meu orientador, prof. Elias Procópio Duarte Jr., por ter sempre me motivado e me auxiliado, mesmo sabendo das minhas limitações com horários para as reuniões; também pelas longas reuniões para discussões do trabalho e revisão dos textos; e, principalmente, por ter sido um grande guia e verdadeiro mestre durante todo o tempo. Agradeço também ao Egon e ao Thiago pelas discussões sobre os problemas de socket, e principalmente ao grande amigo Egon por ter sempre me ajudado quando solicitei qualquer auxílio. Agradeço também ao Luiz Bona pelas discussões sobre os problemas de implementação de timestamp nos seus trabalhos.

Obrigado também à Myrna, por ter estado sempre presente ao meu lado; pela compreensão, carinho e amor, e me dando todo o tempo necessário para que eu pudesse realizar todos os trabalhos que precisava. E estou muito agradecido aos meus pais, Roberto e Vera, que me deram tanto apoio e carinho durante toda a minha vida e que continuam acreditando em mim, por me ajudarem também durante mais este período.

Finalmente, obrigado a todos que me ajudaram de qualquer forma: aos amigos do trabalho, da graduação, do mestrado, e a todos os outros amigos. E obrigado a Deus por permitir mais esta realização na minha vida.

Sumário

Resumo

Este trabalho apresenta um novo modelo genérico de diagnóstico hierárquico adaptativo distribuído e baseado em comparações e um novo algoritmo, chamado *Hi-Dif*, que se baseia neste modelo. O algoritmo permite a detecção de alterações em sistemas com conteúdo replicado distribuído em uma rede como, por exemplo, a Internet. Um nodo sem-falha executando o algoritmo *Hi-Dif*, testa outro nodo do sistema para classificar seu estado. O modelo classifica os nodos do sistema em conjuntos de acordo com o resultado dos testes. Um teste é realizado através do envio de uma tarefa para um par de nodos do sistema. Após o recebimento das duas saídas da execução destas tarefas, o testador compara estas saídas e, se a comparação indicar igualdade, os nodos são classificados no mesmo conjunto. Por outro lado, se a comparação das saídas indicar diferença, os nodos são classificados em conjuntos distintos, de acordo com o resultado da tarefa. Um dos conjuntos contém os nodos sem-falha do sistema. Uma diferença fundamental do modelo proposto para outros modelos publicados anteriormente é que a comparação por um nodo sem-falha, sobre as saídas produzidas por dois nodos falhos, pode resultar em igualdade. Considerando um sistema com N nodos, prova-se que o algoritmo *Hi-Dif* possui latência igual a $log_2 N$ rodadas de testes; o número máximo de testes requeridos pelo algoritmo é de $O(N^2)$ no pior caso; e, que o algoritmo é $(N–1)$-diagnosticável. Resultados experimentais obtidos através de simulações e através de implementação do algoritmo aplicado à Web são apresentados.

Abstract

This work presents a new comparison-based distributed system-level diagnosis generic model and a new algorithm, called *Hi-Dif*, based on this model. The algorithm allows the detection of modifications in systems with replicated data distributed on networks like the Internet. Fault-free nodes running *Hi-Dif*, execute tests on other nodes. Based on test results tested nodes are classified in sets. A test consists of a task that is sent to two system nodes. After the tasks are executed, the task outputs is sent back to the tester. The outputs are then compared; if the comparison produces a match, the two nodes are classified in the same set. On the other hand, if the comparison results in a mismatch, the two nodes are classified in different sets, according to their tasks results. One of the sets always contain all fault-free nodes. One fundamental difference of the proposed model to previously published models is that this model allows the task outputs of two faulty nodes to be equal to each other. Considering a system of N nodes, it is proved that the algorithm has latency equal to $log_2 N$ testing rounds; the maximun number of tests required for the algorithm is $O(N^2)$ in the worst case; and, the algorithm is $(N-1)$-diagnosable. Experimental results obtained by simulation and by the implementation of the algorithm applied to the Web are presented.

Capítulo 1

Introdução

Atualmente já se aproxima de 600 milhões o número estimado de pessoas que utilizam a Internet [1]. O bom funcionamento da rede é cada vez mais importante para indivíduos e organizações. Por outro lado, ataques e ações de vandalismo como, por exemplo, modificações não autorizadas de conteúdo na Web têm se tornado cada vez mais comuns [2]. Com isso, cresce a preocupação com a monitoração de sistemas visando a detecção de violações.

O objetivo de um sistema de monitoração é descobrir quais são as unidades falhas de um sistema. É essencial que a monitoração seja tolerante a falhas, ou seja, capaz de funcionar corretamente mesmo na presença de unidades falhas [3]. Este trabalho trata da monitoração, utilizando diagnóstico distribuído baseado em comparações, para sistemas com conteúdo replicado em uma rede como, por exemplo, a Internet. Uma das aplicações práticas deste trabalho é o diagnóstico de vandalismo em servidores Web com dados replicados [4].

A seguir, neste capítulo, é apresentada uma visão geral sobre a detecção de alterações e trabalhos relacionados, em seguida, é feita uma introdução ao diagnóstico

distribuído baseado em comparações e, na sequência, algoritmos de diagnóstico hierárquico e adaptativo e o algoritmo proposto são descritos brevemente.

1.1 Detecção de Alterações: Trabalhos Relacionados

Estatísticas mostram que atualmente o número de sites invadidos na Internet é muito grande [2, 5]. Por exemplo, segundo notícias de um serviço especializado em registrar invasões de sites, em 2001 o número de sites invadidos e vandalizados virtualmente, ou seja, com conteúdo modificado, em todo o mundo foi de 22,4 mil [5, 6]. Em 2002, apenas no primeiro semestre do ano, estatísticas já mostravam que o número de sites invadidos e vandalizados já chegavam a 20.371 [7], 27% a mais em relação ao mesmo período de 2001 [7].

Com o objetivo de minimizar as chances de um intruso obter sucesso em suas atividades, diversos mecanismos de proteção podem ser utilizados. Entre estes mecanismos surgem a criptografia, a certificação digital, a infra-estrutura de chaves públicas e privadas, os *firewalls*, os protocolos de autenticação, os sistemas de detecção de intrusão ou IDSs (*Intrusion Detection Systems*), e também os sistemas para monitoramento e detecção de alterações [8, 9].

Existem muitas ferramentas e trabalhos publicados para detecção de alterações, como a detecção de modificações ou atualizações em uma página Web que é frequentemente acessada [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Muitas ferramentas permitem até monitorar as atualizações em partes específicas da página [10, 11, 12, 14, 16, 17, 20]. Estas ferramentas utilizam comparações para poder determinar quando

houve alguma alteração. Muitas destas ferramentas armazenam uma cópia local dos arquivos [10] ou algumas armazenam um *check-sum* dos arquivos [13] que estão sendo monitorados para se poder comparar com o arquivo ou com a parte de interesse do arquivo que está na Web. Um exemplo que funciona como este modelo é a ferramenta WebMon [10]. Esta ferramenta configura um conjunto de páginas Web que serão monitoradas e armazena uma cópia local dos arquivos. Através de um intervalo configurado, as páginas Web são consultadas para a verificação de diferenças com relação às cópias locais armazenadas. Outras ferramentas ainda executam como um serviço na própria Web e enviam um e-mail quando detectam a alteração de uma página Web, como por exemplo a ferramenta URL-minder [13].

O modelo proposto neste trabalho é distribuído, ou seja, não há um elemento centralizado responsável pela monitoração. Ao contrário, as unidades – que possuem os dados replicados – se monitoram de acordo com um modelo distribuído de diagnóstico baseado em comparações, que é tópico das seções seguintes deste capítulo.

1.2 Diagnóstico Distribuído Baseado em Comparações

O objetivo do diagnóstico em nível de sistema é identificar quais unidades do sistema estão falhas e quais estão sem-falhas. As unidades do sistema realizam testes sobre as outras unidades e comunicam resultados de testes visando obter o diagnóstico completo do sistema.

Diferentes modelos de diagnóstico em nível de sistema [21] têm sido propostos. O primeiro modelo de diagnóstico foi o modelo PMC (Preparata, Metze, e Chien) [22].

Este modelo assume que as unidades têm a capacidade de testarem outras unidades. Uma unidade pode estar em um de dois estados, *falho* ou *sem-falha*, e seu estado não muda durante o diagnóstico. Neste modelo, um teste envolve a aplicação controlada de estímulos e a observação da resposta correspondente. O modelo também assume que todo nodo sem-falha executa de modo confiável todos os testes, isto é, um nodo sem-falha nunca realiza um teste de forma imprecisa. Uma unidade testadora avalia a unidade testada como falha ou sem-falha. O conjunto de todos os resultados de testes é chamado de síndrome do sistema. O modelo PMC assume a existência de um observador central que, baseado na síndrome, realiza o diagnóstico.

Muitos algoritmos baseados no modelo PMC, bem como variações deste modelo, têm sido propostos nas últimas décadas. Se cada nodo do sistema é capaz de executar testes em outros nodos e se os resultados dos testes dependem dos resultados de testes anteriores, então o diagnóstico é dito adaptativo [23]. Se o modelo ou algoritmo não assume a existência de um nodo monitor, mas considera que os próprios nodos que realizam os testes fazem o diagnóstico do sistema então o diagnóstico é dito distribuído [24]. Quando o diagnóstico é hierárquico, os nodos são divididos em clusters e a cada rodada de testes o tamanho destes clusters aumenta [25].

Em um algoritmo de diagnóstico hierárquico, distribuído e adaptativo, quando o nodo testador realiza um teste em um nodo sem-falha, recebe informações do nodo testado que são chamadas de informações de diagnóstico [26]. Os algoritmos hierárquicos, distribuídos e adaptativos baseados no modelo PMC assumem que um nodo com falha simplesmente pára e nunca responde a testes [27, 28, 29].

Os modelos de diagnóstico baseados em comparações foram propostos inicialmente por Malek [30] e em seguida por Chwa e Hakimi [31]. Nestes modelos, é assumido que em um sistema de *N* unidades, a comparação dos resultados produzidos na saída da execução de tarefas de alguns ou todo par de unidades é possível. Qualquer diferença na comparação indica que uma ou ambas as unidades estão falhas. Estes modelos também assumem que existe um observador central que armazena as informações das saídas de tarefas e que através das comparações das saídas chega-se ao diagnóstico completo do sistema: se existe alguma falha ou não, e/ou localiza as unidades falhas. Este observador central é uma unidade confiável que não sofre nenhum evento. A diferença deste modelo proposto em [31] para o modelo proposto por Malek [30], é que duas unidades falhas, quando submetidas à mesma tarefa, podem produzir as mesmas saídas, podendo assim, causar uma igualdade na comparação destas saídas.

O modelo MM (Maeng e Malek) [32], é uma extensão do modelo de diagnóstico baseado em comparação proposto inicialmente por Malek [30]. Este modelo assume que os resultados das comparações das saídas são enviados para um observador central que realiza o diagnóstico completo do sistema, mas permite que as comparações das saídas das tarefas sejam feitas pelas próprias unidades. A única restrição é que a unidade que realiza a comparação deve ser diferente das duas unidades que produzem as saídas da tarefa executada. Sengupta e Dahbura em [33] propuseram uma generalização do modelo MM chamada de modelo MM*. Este modelo permite que a própria unidade testadora seja uma das unidades que produzem as saídas.

Blough e Brown em [34] apresentaram um modelo de diagnóstico baseado em comparações com broadcast confiável. Neste modelo, uma tarefa é atribuída como

entrada para um par de nodos diferentes. Estes dois nodos executam a tarefa e as saídas produzidas são comparadas para detectar uma possível falha em um dos dois nodos. Estes dois nodos que estão sendo comparados fazem broadcast das suas saídas para todos os nodos do sistema, inclusive eles próprios. As comparações das saídas são realizadas em todo nodo sem-falha do sistema.

Em [35] Wang propõe um modelo de diagnóstico baseado em comparações utilizando hipercubos [36, 37] e também nos chamados hipercubos melhorados (*enhanced hypercubes*) [38]. Cada nodo do sistema realiza os testes em outros nodos do sistema através da comparação das saídas de tarefas atribuídas aos nodos que estão sendo comparados. Este modelo também permite que o próprio nodo testador possa ser um dos nodos que estão sendo comparados.

Araki e Shibata em [39] estudam a diagnosticabilidade de redes borboletas [40] através do diagnóstico baseado em comparações. São propostos dois métodos de geração de testes em redes borboletas, um é chamado de comparação *one-way*, e o outro de comparação *two-way*. O diagnóstico é realizado através do envio da mesma tarefa de entrada para um par de processadores. Para isso é realizada a comparação das respostas destas tarefas por um terceiro processador que tem acesso a ambos os processadores que estão sendo comparados. O conjunto com o resultado de todas as comparações também é chamado de síndrome do sistema. Também considerando o diagnóstico baseado em comparações, Fan em [41] avalia a diagnosticabilidade de cubos cruzados. Neste modelo o diagnóstico do sistema também é obtido através da síndrome do sistema.

1.3 Diagnóstico Hierárquico e Adaptativo

Para os algoritmos de diagnóstico hierárquico, adaptativo e distribuído baseados no modelo PMC, um nodo pode estar em um de dois estados, falho ou sem-falha, e seu estado não muda durante o diagnóstico. Estes algoritmos definem rodadas de testes, que são intervalos de tempo em que cada nodo executa seus testes. Quando o nodo testador realiza um teste em um nodo sem-falha, recebe informações do nodo testado que são chamadas de informações de diagnóstico [26].

Os algoritmos hierárquicos adaptativos e distribuídos incluem o algoritmo *Hi-ADSD* [26], o algoritmo *Hi-ADSD with Detours* [42] e por último o *Hi-ADSD with Timestamps* apresentado em [29]. Estes algoritmos utilizam uma estratégia de agrupar os nodos em clusters lógicos de tamanho progressivo, com exceção do algoritmo *Hi-ADSD with Timestamps* no qual o tamanho do cluster, considerando um sistema com N nodos, é sempre $N/2$. O número de nodos em um cluster, isto é, seu tamanho, é sempre uma potência de dois. No algoritmo *Hi-ADSD with Timestamps*, cada nodo do sistema executando este algoritmo, mantém um *timestamp* para o estado de cada outro nodo do sistema. Desta forma o testador pode obter apenas a informação de diagnóstico mais recente sobre o estado de todos os nodos do sistema. Como exemplo, a figura 1.1 mostra um sistema de 8 nodos; os arcos correspondem aos testes executados, desta forma, há um arco do nodo i para o nodo j, se o nodo i testa o nodo j. Esta figura forma um hipercubo [36, 37] quando todos os nodos do sistema estão sem-falha. Considerando este sistema de 8 nodos executando o algoritmo *Hi-ADSD with Timestamps*, a figura 1.2 mostra os clusters testados pelo nodo 0. Por exemplo, quando o nodo 0 testa o nodo 1, o

nodo 0 obtém informações sobre os 4 nodos (*N*/2 nodos), ou seja, sobre os nodos 1, 3, 5 e 7. Quando o nodo 0 testa o nodo 2, o nodo 0 obtém informações sobre os 4 nodos 2, 3, 6 e 7. Quando o nodo 0 testa o nodo 4, o nodo 0 obtém informações sobre os nodos 4, 5, 6 e 7.

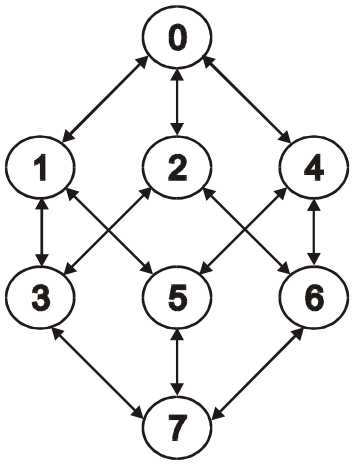

Figura 1.1: Sistema de 8 nodos quando todos os nodos estão sem-falha.

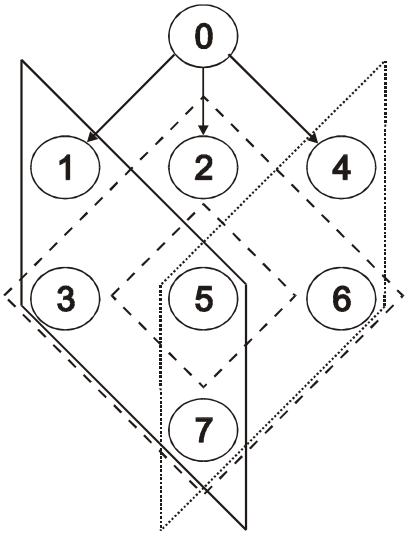

Figura 1.2: Clusters testados pelo nodo 0 em um sistema de 8 nodos.

O modelo genérico de diagnóstico distribuído e adaptativo em nível de sistema baseado em comparações é apresentado por Albini e Duarte em [43]. Este modelo usa a mesma estratégia hierárquica de testes similar à implementada no algoritmo *Hi-ADSD with Timestamps*, mas não é limitado a falhas do tipo crash como os algoritmos baseados no modelo PMC. Um nodo sem-falha testa outro nodo do sistema para identificar seu estado. Um teste é realizado através do envio de uma tarefa para um par de nodos do sistema. Após o recebimento das duas saídas da execução destas tarefas, o testador compara estas saídas e se a comparação indicar igualdade, os nodos são considerados sem-falha. Este modelo possui a seguinte asserção: quando um nodo sem-falha compara as saídas produzidas por dois nodos falhos, o resultado sempre indica diferença. Neste trabalho apresentado em [43], é apresentado o algoritmo *Hi-Comp* (*Hierarchical Comparison-Based Adaptive Distributed System-Level Diagnosis Algorithm*), proposto para o modelo genérico de diagnóstico baseado em comparações ali definido.

1.4 O Novo Algoritmo Proposto

Este trabalho propõe um novo modelo genérico para o diagnóstico distribuído e adaptativo em nível de sistema baseado em comparações. Os nodos do sistema podem estar falhos ou sem-falhas. Um nodo falho pode tanto estar com falha do tipo crash, como apenas estar respondendo de maneira incorreta à tarefa enviada como teste.

Este novo modelo usa a mesma estratégia hierárquica de testes implementada no algoritmo *Hi-ADSD with Timestamps* [29], mas também não é limitado a falhas do tipo crash. Um nodo sem-falha testa outro nodo do sistema para classificar seu estado. O

modelo classifica os nodos do sistema em conjuntos de acordo com o resultado dos testes. Um teste é realizado através do envio de uma tarefa para um par de nodos do sistema. Após o recebimento das duas saídas da execução destas tarefas, o testador compara estas saídas e, se a comparação indicar igualdade, os nodos são classificados no mesmo conjunto de nodos. Por outro lado, se a comparação das saídas indicar diferença, os nodos são classificados em conjuntos distintos, de acordo com o resultado da tarefa. Um dos conjuntos contém os nodos sem-falha do sistema.

Uma diferença fundamental do modelo proposto para outros modelos publicados anteriormente, é que a asserção – a comparação, realizada por um nodo sem-falha, das saídas produzidas por dois nodos falhos sempre resulta em diferença – foi eliminada. Desta forma, este modelo permite o tratamento da situação onde dois nodos quaisquer estão invadidos e vandalizados com as mesmas alterações sobre o conteúdo replicado.

Este trabalho apresenta o algoritmo *Hi-Dif* baseado no modelo proposto para o diagnóstico adaptativo distribuído e hierárquico baseado em comparações. Este novo algoritmo é voltado para o diagnóstico de alterações em nodos com conteúdo replicado, como, por exemplo, o diagnóstico de vandalismo em servidores Web com dados replicados. Este novo algoritmo ainda permite a identificação e diferenciação dos nodos que estão com seu conteúdo modificado, ou seja, permite a identificação dos nodos que possuem a mesma modificação de conteúdo e diferenciá-los de outros nodos que possuem outra modificação.

Uma rodada de teste é um intervalo de tempo onde todos os nodos sem-falha do sistema obtêm informação de diagnóstico sobre todos os nodos do sistema. Dada esta

definição e a forma com que os testes são realizados, o novo algoritmo possui uma latência igual a $log_2 N$ rodadas de testes para um sistema de N nodos. Também prova-se que o número máximo de testes requeridos pelo algoritmo é de $O(N^2)$ no pior caso, e que o algoritmo é $(N–1)$-diagnosticável.

Resultados experimentais obtidos através de simulações do algoritmo e através de implementação do algoritmo – utilizada para o diagnóstico de nodos com conteúdo replicado na Web – são apresentados.

1.5 Organização deste Trabalho

O restante deste trabalho está organizado da seguinte maneira. O capítulo 2 apresenta o diagnóstico distribuído baseado em comparações. Em seguida, o capítulo 3 descreve o novo modelo e o novo algoritmo de diagnóstico distribuído baseado em comparações. No capítulo 4 são apresentados os resultados experimentais obtidos através de simulações do algoritmo. Em seguida, no capítulo 5 são descritos os resultados experimentais através de implementação do algoritmo utilizada para o diagnóstico de nodos com conteúdo replicado na Web. E, no capítulo 6 seguem as conclusões e trabalhos futuros.

Capítulo 2

Diagnóstico Distribuído Baseado em Comparações

O diagnóstico em nível de sistema permite que os componentes sem-falha de um sistema distribuído determinem quais unidades estão falhas e quais estão sem-falhas [25]. Se o algoritmo considera que os próprios nodos que realizam os testes fazem o diagnóstico do sistema então o algoritmo é dito distribuído [24].

Muitos resultados sobre diagnóstico distribuído em nível de sistema têm sido publicados e muitos modelos têm sido apresentados nas últimas décadas. Este capítulo apresenta alguns destes modelos, incluindo o modelo PMC, modelos distribuídos de diagnóstico hierárquico e adaptativo e também modelos baseados em comparações.

2.1 Modelo PMC

O primeiro modelo proposto para diagnóstico em nível de sistema foi o modelo PMC [22], cujo nome se refere às iniciais dos autores do modelo, Preparata, Metze e

Chien. No modelo PMC, o diagnóstico do sistema assume que as unidades têm a capacidade de testarem outras unidades. Uma unidade pode estar em um de dois estados, *falho* ou *sem-falha*, e seu estado não muda durante o diagnóstico. Neste modelo, um teste envolve a aplicação controlada de estímulos e a observação da resposta correspondente.

O modelo PMC assume que um sistema S é representável por um grafo completo, isto é, há ligação entre quaisquer pares de unidades (u_i, u_j) e não existem falhas de enlace. O modelo também assume que todo nodo sem-falha executa de modo confiável todos os testes, isto é, um nodo sem-falha nunca realiza um teste de forma imprecisa, enquanto nodos com falha podem retornar resultados incorretos [44, 25, 22]. Uma unidade testadora avalia a unidade testada como falha ou sem-falha.

O conjunto de testes deste modelo forma um grafo direcionado em que os vértices ou unidades u_i são os nodos do sistema e as arestas que vão de i para j, ou links de testes b_{ij}, representam um teste do nodo i para o nodo j. O valor a_{ij} que pode ser $\{0, 1\}$, corresponde ao resultado do teste b_{ij}. Este resultado será 0 sobre a hipótese de que o testador u_i está sem-falha e se a unidade testada u_j estiver sem-falha. O resultado será 1 também considerando a mesma hipótese e se u_j estiver falha. Mas caso o testador u_i esteja falho, o resultado não é confiável, então a_{ij} pode assumir tanto 0 como 1, independente do estado da unidade testada.

O conjunto de todos os resultados de testes é chamado de síndrome do sistema. O modelo PMC assume a existência de um observador central que, baseado na síndrome, realiza o diagnóstico, isto é, determina o estado de todos os nodos do sistema.

Hakimi e Amim em [44] apresentam resultados sobre a capacidade de realização de diagnóstico do modelo PMC, tendo em vista o número de nodos falhos no sistema. Se o sistema consiste de N unidades, as seguintes condições garantem que o diagnóstico é possível mesmo que até t nodos estejam falhos: 1) $N \geq 2t+1$; e 2) cada unidade é testada por pelo menos outras t unidades.

Como exemplo, a figura 2.1 mostra um sistema com 5 unidades u_1, u_2, u_3, u_4 e u_5 executando os testes b_{12}, b_{23}, b_{34}, b_{45} e b_{51} onde a unidade 1 é falha. Nesta figura, ao lado das arestas também é mostrado o valor a_{ij} correspondente. A síndrome para o sistema mostrado nesta figura é representada por um vetor de 5 bits (x, 0, 0, 0, 1) que correspondem aos valores (a_{12}, a_{23}, a_{34}, a_{45}, a_{51}), onde x pode ser tanto 0 como 1.

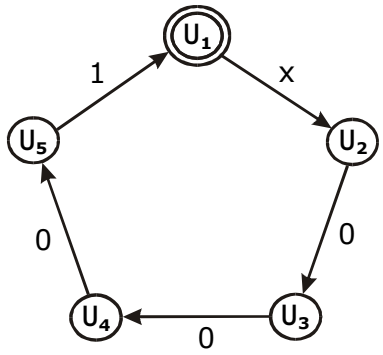

Figura 2.1: Um sistema com 5 onde a unidade 1 é falha.

2.2 Diagnóstico Hierárquico e Adaptativo

Muitos algoritmos baseados no modelo PMC foram propostos nos últimos 35 anos. O diagnóstico é dito adaptativo [23], se cada nodo do sistema é capaz de executar testes em outros nodos e se a seleção dos próximos testes depende dos resultados de

testes anteriores. O diagnóstico é dito distribuído [24], se o algoritmo não assume a existência de um nodo observador central, mas considera que os próprios nodos que realizam os testes fazem o diagnóstico do sistema. Quando o diagnóstico é hierárquico [25], os nodos são divididos em clusters e a cada rodada de testes o tamanho destes clusters aumenta. Um nodo pode estar em um de dois estados, falho ou sem-falha, e seu estado não muda durante o diagnóstico. Cada mudança no estado dos nodos é chamada de evento.

Os algoritmos hierárquicos, adaptativos e distribuídos assumem que o sistema *S* com *N* nodos é representável por um grafo completo, isto é, há ligação entre quaisquer pares de nodos (n_i, n_j) e não existem falhas de enlace. Estes algoritmos definem rodadas de testes, que são intervalos de tempo em que cada nodo executa seus testes. Quando o nodo testador realiza um teste em um nodo sem-falha, recebe informações do nodo testado que são chamadas de informações de diagnóstico [25]. Os algoritmos hierárquicos, adaptativos e distribuídos assumem falhas do tipo crash, isto é, um nodo com falha simplesmente pára e nunca responde a testes [27, 28, 29].

Os algoritmos hierárquicos adaptativos e distribuídos incluem o algoritmo *Hi-ADSD* [26], o algoritmo *Hi-ADSD with Detours* [42] e por último o *Hi-ADSD with Timestamps* [29]. Estes algoritmos utilizam uma estratégia de agrupar os nodos em clusters lógicos de tamanho progressivo, com exceção do algoritmo *Hi-ADSD with Timestamps* onde o tamanho do cluster, considerando um sistema com *N* nodos, é sempre *N*/2. O número de nodos em um cluster, isto é, seu tamanho, é sempre uma potência de dois. Estes algoritmos executam seus testes de maneira assíncrona, isto é,

em um determinado intervalo de testes, os nodos sem-falha podem testar clusters de tamanhos diferentes.

2.2.1 O Algoritmo *Hi-ADSD with Timestamps*

O algoritmo *Hi-ADSD with Timestamps* foi proposto por Duarte, Brawerman e Albini [29]. Este algoritmo utiliza a mesma estratégia de agrupar os nodos em clusters lógicos utilizada pelos outros algoritmos hierárquicos. Neste algoritmo o tamanho de todo cluster é sempre $N/2$. O grafo dirigido dos nodos sem-falhas testados, $T(S)$, tem arestas dirigidas do nodo *a* para o nodo *b* caso o nodo *a* teste o nodo *b* como sem-falha. Quando não há falhas em nenhum nodo do sistema, $T(S)$ é um hipercubo, como mostra a figura 2.2 para $N=8$.

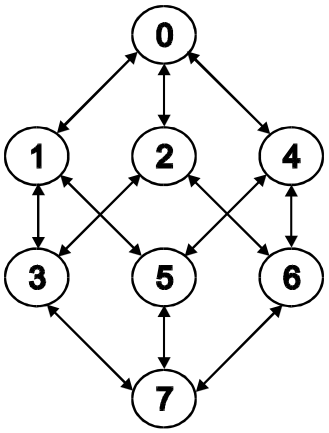

Figura 2.2: $T(S)$ para um sistema com 8 nodos.

Quando um nodo é testado, o nodo testador obtém informação sobre todo o cluster do nodo testado. Em cada intervalo de teste, cada nodo testa um determinado nodo em um cluster. Em $log_2 N$ intervalos de testes todos os clusters são testados. O processo continua indefinidamente.

No algoritmo *Hi-ADSD with Timestamps* é possível que o nodo *i* obtenha informação de diagnóstico sobre o nodo *j* a partir de dois ou mais nodos sem-falha. Neste caso é necessário garantir que o nodo *i* seja capaz de determinar a informação mais recente sobre o estado do nodo *j*. Por exemplo, na figura 2.3 o nodo 0 obtém informação de diagnóstico do nodo 5 através do nodo 1 e do nodo 4. Consequentemente é preciso garantir que o nodo 0 obtenha a informação mais recente sobre o nodo 5.

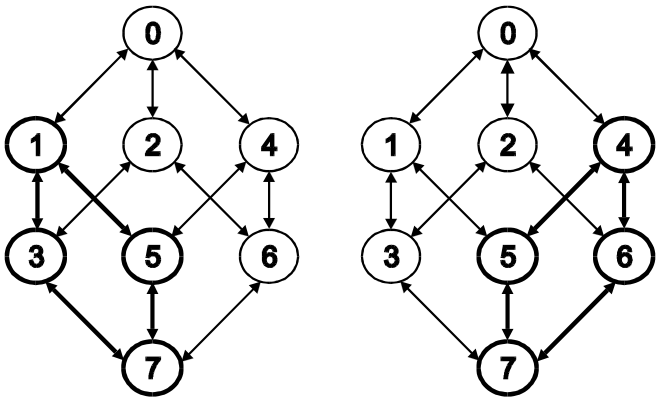

Figura 2.3: O nodo 0 obtém informação de diagnóstico do nodo 5 através do nodo 1 e do nodo 4.

Assim, cada nodo do sistema executando o algoritmo *Hi-ADSD with Timestamps* mantém um *timestamp* para o estado de cada outro nodo do sistema, de forma que o testador possa determinar qual é a informação de diagnóstico mais recente sobre certo nodo obtida a partir de mais de um nodo testado. A latência deste algoritmo é $log^2{}_2 N$ rodadas de testes e o número de testes por rodada de testes é de $N log_2 N$ e a cada rodada de testes é transferido um número fixo de informações de diagnóstico de $N/2$ nodos a cada teste.

2.3 Diagnóstico Baseado em Comparações

O primeiro modelo de diagnóstico baseado em comparações foi proposto por Malek [30]. Neste modelo, é assumido que em um sistema de *N* unidades, a comparação dos resultados produzidos na saída da execução de tarefas de alguns ou todo par de unidades é possível. Qualquer diferença na comparação indica que uma ou ambas as unidades estão falhas. Além disso, ele consiste de duas atividades: primeiro, detecção de falhas, onde o objetivo é somente determinar que uma falha ocorreu em um sistema, pois se a comparação de resultados indicar diferença sabe-se que ao menos uma das duas unidades estão falhas, mas não se sabe qual; e segundo, localização de falhas, onde a identificação de qual é a unidade falha é necessária.

Um sistema com *N* unidades é modelado como um grafo *G*(*V*, *E*, *C*) onde *V* é o conjunto de *N* vértices e *E* e *C* são conjuntos de arestas. Cada componente de *V* corresponde a um processador que é considerado uma unidade do sistema e cada elemento do conjunto *E* representa a comunicação entre as unidades. O conjunto *C* corresponde às comparações que são um subconjunto de *E*. Existe uma aresta de comparação, se o resultado das tarefas executadas pelas entidades que são os vértices desta aresta, forem comparados por uma entidade testadora.

O modelo assume que as tarefas do sistema são duplicadas em duas unidades diferentes. Ele também assume que existe um observador central que armazena as informações das saídas de tarefas e que através das comparações das saídas chega-se ao diagnóstico completo do sistema: se existe alguma falha ou não, e/ou localiza as

unidades falhas. Este observador central é uma unidade confiável que não sofre nenhum evento.

Quando duas unidades são comparadas, os resultados possíveis são mostrados na figura 2.4. O resultado *pass* indica que ambas as unidades estão sem-falha, enquanto *fail* indica que pelo menos uma das duas unidades está falha. Para a determinação da existência de falhas, mais comparações com relação a este par de unidades comparadas serão redundantes, mas para a localização de falhas, as unidades podem ser comparadas com outras unidades com o objetivo de identificar qual ou quais as unidades estão falhas.

Unidade 1	**Unidade 2**	**Comparação das Saídas**
sem-falha	sem-falha	0 *(pass)*
sem-falha	falha	1 *(fail)*
falha	sem-falha	1 *(fail)*
falha	falha	1 *(fail)*

Figura 2.4: O estado e a comparação das saídas de um par de unidades.

Introduzido juntamente com este primeiro modelo [30], foi proposto por Chwa e Hakimi em [31] outro modelo de diagnóstico baseado em comparações. Neste modelo, também são enviadas tarefas replicadas para duas unidades do sistema. Os estados falho ou sem-falha das unidades são determinados pela comparação das saídas destas tarefas. Uma diferença na comparação das saídas destas tarefas indica a existência de uma falha. Também é assumida a existência de um observador central que realizada estas comparações para obter o diagnóstico completo do sistema. A diferença deste modelo para o modelo proposto por Malek, é que duas unidades falhas, quando submetidas à

mesma tarefa, podem produzir as mesmas saídas, podendo assim, causar uma igualdade na comparação destas saídas.

2.4 Modelo MM de Diagnóstico Baseado em Comparações

O modelo MM, das iniciais dos autores, Maeng e Malek [32], é uma extensão do modelo de diagnóstico baseado em comparação proposto inicialmente por Malek [30]. No modelo apresentado na seção 2.3 o sistema pode ser representado por um grafo $G=(V, E)$ não direcionado, onde cada nodo representa um processador e cada aresta representa um enlace de comunicação entre cada par de processadores. Dois processadores interagem um com o outro através do envio de mensagens sobre este link de comunicação. A estratégia do diagnóstico pode ser modelada por um multigrafo $M=(V, C)$ definido sobre o mesmo conjunto de nodos do grafo G. Uma aresta $(i, j) \in C$ representa o fato que as unidades $i, j \in V$ podem ser comparadas. M é um multigrafo porque as saídas do mesmo par de unidades podem ser comparadas por muitas outras diferentes unidades do sistema.

Neste modelo, a identificação do estado falho ou sem-falha de uma unidade é determinada pela comparação da resposta de uma tarefa do sistema com a resposta da mesma tarefa imposta a outras unidades do sistema. Uma unidade k compara a saída produzida por duas unidades i e j em resposta à mesma tarefa de entrada, uma discordância no resultado da comparação indica que ao menos uma unidade está falha e esta comparação resulta 1, enquanto que uma igualdade na comparação indica que as duas unidades estão sem-falha e esta comparação resulta 0.

O conjunto de todas as comparações resultantes é nomeado de síndrome do sistema. Em [31] e [30] é assumido que um observador central do sistema realiza as comparações para descobrir quais são as unidades falhas do sistema. Este modelo assume que os resultados das comparações das saídas são enviados para um observador central que realiza o diagnóstico completo do sistema, mas permite que as comparações das saídas das tarefas sejam feitas pelas próprias unidades. A única restrição é que a unidade que realiza a comparação deve ser diferente das duas unidades que produzem as saídas da tarefa executada.

Este modelo assume algumas asserções como: todas as falhas de qualquer unidade são permanentes; uma unidade que está falha produz saídas incorretas para toda tarefa de entrada; a comparação feita por uma unidade falha não é confiável; duas unidades falhas, quando submetidas à mesma tarefa de entrada não produzem a mesma saída; e, existe um limite superior *t*, sobre a quantidade de unidades que podem estar falhas no sistema para que a síndrome deste sistema seja possível.

A figura 2.5(a) mostra um sistema modelado como um grafo *G*, e a figura 2.5(b) mostra o conjunto de comparações para um multigrafo *M* referente a este grafo *G*. Nesta figura, o número ao lado de cada aresta é o número da unidade que está comparando o resultado das saídas do par de unidades dos vértices desta aresta. O resultado de cada comparação é mostrado dentro de um círculo ao lado de cada uma destas arestas. Nesta figura podemos notar que a síndrome do sistema pode ser produzida, por exemplo, quando a unidade 2 for a unidade falha.

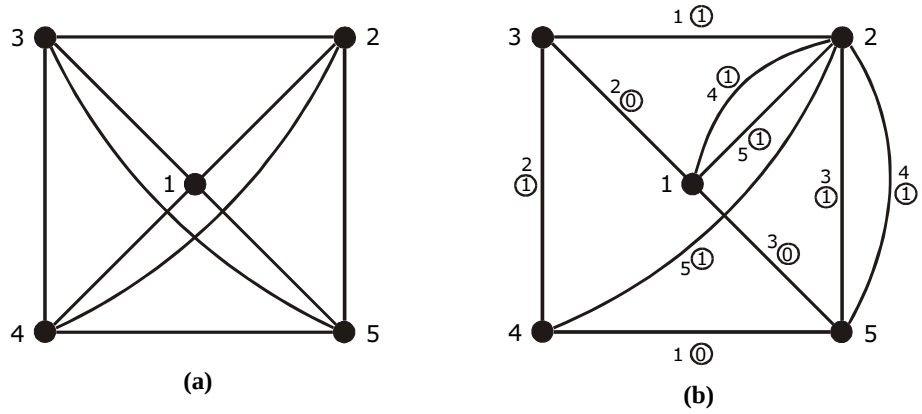

Figura 2.5: (a) Um sistema para um grafo *G* de 5 unidades.
(b) Um multigrafo *M* de comparações para o grafo *G*.

Uma generalização do modelo MM foi proposta por [33] e chamada de modelo MM*. Neste modelo as comparações são distribuídas e as saídas para comparações são enviadas para um observador central que realiza o diagnóstico. Mas este modelo permite que a própria unidade testadora seja uma das unidades que produzem as saídas. Este modelo também apresenta uma solução para a classe de sistemas diagnosticáveis pelo modelo MM e dá as condições suficientes para um sistema ser *t*-diagnosticável.

2.5 Modelo de Diagnóstico Baseado em Comparações com Broadcast Confiável

O modelo de diagnóstico baseado em comparações com broadcast confiável [34] é uma combinação do diagnóstico distribuído [45] e o modelo MM* de comparação em sistemas [33] utilizando um serviço de broadcast confiável [46]. O diagnóstico distribuído é baseado na comparação de tarefas redundantes e tem acesso ao protocolo de broadcast confiável. Neste modelo, uma tarefa é atribuída como entrada para um par de nodos diferentes. Estes dois nodos executam a tarefa e as saídas produzidas são

comparadas para detectar uma possível falha em um dos dois nodos. Estes dois nodos que estão sendo comparados fazem broadcast das suas saídas para todos os nodos do sistema, inclusive eles próprios. A figura 2.6 mostra este processo. Nesta figura o nodo 1 envia uma mesma tarefa aos nodos 2 e 3 que fazem broadcast das saídas destas tarefas para todos os nodos do sistema.

Todo nodo sem-falha no sistema compara as duas saídas que são recebidas. As comparações são realizadas em todo nodo sem-falha, inclusive os próprios nodos que estão sendo comparados. Uma vez que um nodo produziu um conjunto suficiente de comparações de saída sobre os diversos nodos do sistema, ele é responsável por diagnosticar o estado do sistema assumindo que ele próprio é um nodo sem-falha.

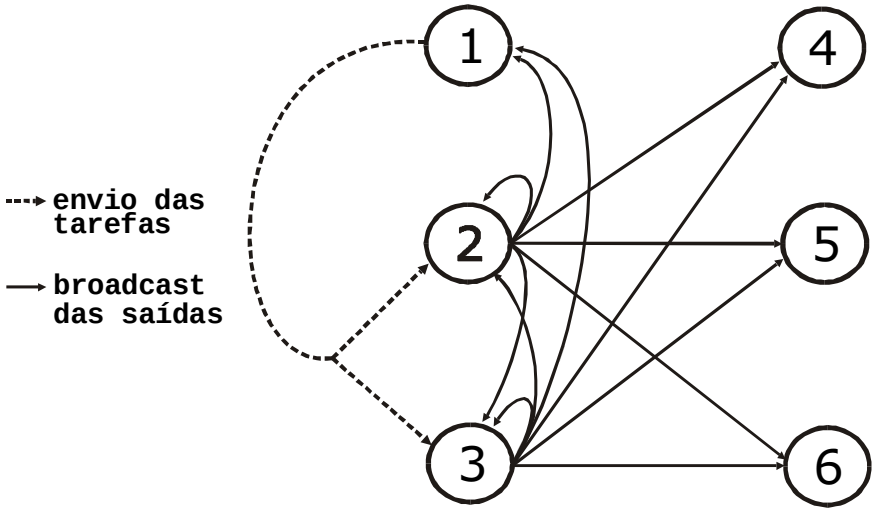

Figura 2.6: Em um sistema de 6 nodos, uma tarefa é enviada para os nodos 2 e 3 e suas saídas são enviadas por broadcast para todos os nodos do sistema.

Este modelo adota as mesmas asserções sobre as comparações das saídas usadas originalmente pelo modelo de Malek [30], pelo modelo MM [32] além do modelo MM* [33]. Estas asserções mostram que dois nodos sem-falha submetidos à mesma tarefa

sempre produzem a mesma saída, enquanto que um nodo falho sempre produz uma saída diferente da saída de qualquer outro nodo falho ou sem-falha submetido à esta mesma tarefa. Esta característica destes modelos difere-se para o modelo baseado em comparação de Chwa e Hakimi [31] em que dois nodos falhos podem produzir saídas iguais quando submetidos às mesmas tarefas.

A comparação dos nodos deste sistema também é modelada como um grafo $G(V, E)$, onde V é um conjunto de vértices e E é um conjunto de arestas. Cada vértice corresponde a um nodo e cada aresta corresponde a um par de nodos que serão comparados durante o procedimento de diagnóstico.

Se as saídas das tarefas de um par de nodos são iguais, a comparação resulta 0. Se as saídas das tarefas de dois pares de nodos são diferentes, a comparação resulta 1. Uma síndrome é a coleção completa das comparações de todas as saídas, isto é, um valor 0 ou 1 para cada aresta de comparação do grafo G. Caso aconteça um *time-out*, ou seja, um tempo limitado é esgotado sem uma resposta com a saída de uma certa tarefa, é utilizado um valor especifico que é diferente quando comparado com qualquer outro valor.

Além disso, um sistema utilizando um modelo baseado em comparações com broadcast precisa garantir outras asserções: qualquer mensagem de broadcast de um nodo sem-falha é recebida corretamente por todos os outros nodos sem-falha em um tempo limitado; o tempo para qualquer mensagem produzir uma saída também é limitado; cada nodo tem um identificador único; nodos sem-falha podem identificar corretamente o remetente de um broadcast; e, valores de saída enviados por um nodo

falho e recebidos por nodos sem-falha não causam um resultado igual quando comparados com a saída de um outro nodo qualquer falho ou sem-falha.

Por fim, o broadcast confiável utilizado, é o tipo mais fraco possível dentre todos os tipos de broadcast confiável [46], e não tem requisitos quanto à ordem das mensagens, isto é, não é um broadcast FIFO (*Fisrt In, Fist Out*) e nem é um broadcast causal [46]. Estas características ajudaram a simplificar o problema deste modelo, garantindo que todos os nodos sem-falha do sistema produzem a mesma saída para as comparações e uma síndrome completa é obtida para cada nodo sem-falha do sistema.

2.6 Modelo de Diagnóstico Baseado em Comparações Utilizando Hipercubos

O modelo de diagnóstico baseado em comparações utilizando hipercubos foi proposto por Wang [35] e trata do diagnóstico baseado em comparações em hipercubos e nos chamados hipercubos melhorados (*enhanced hypercubes*) [38]. Considere um grafo $G=(V, E)$, onde cada nodo $v_i \in V$ representa um processador e cada aresta $\{v_i, v_j\} \in E$ representa um enlace de comunicação entre v_i e v_j. O sistema pode ser modelado como um multigrafo $M=(V, C)$ onde V representa o conjunto de processadores e as arestas $\{v_b, v_c\}_{v_a} \in C$, onde a aresta com label v_a conectando os nodos v_b e v_c, representa que estes nodos v_b e v_c estão sendo comparadas pelo nodo v_a. M é um multigrafo porque o mesmo par de nodos pode ser comparado por diferentes nodos do sistema. Este modelo define sobre o multigrafo M, um multigrafo $M^*=(V, C^*)$, onde $C^* = \{\{v_b, v_c\}_{v_a} \mid \{v_a, v_b\}, \{v_b, v_c\} \in E\}$, ou seja, este modelo permite que o

próprio nodo testador possa ser um dos nodos que estão sendo comparados. Um grafo *G* para um sistema de 5 nodos e um multigrafo *M* são exemplificados na figura 2.7. Por exemplo, nesta figura através do multigrafo *M*, podemos notar que os nodos v_a e v_e são comparados pelo nodo v_b, os nodos v_a e v_b são comparados pelo nodo v_e, os nodos v_b e v_e são comparados pelo nodo v_a, e assim por diante.

Para representar o resultado da comparação dos nodos v_b e v_c pelo nodo v_a, é usada a notação $r(\{v_b, v_c\}_{v_a})$. Este resultado será 0 quando a comparação das saídas indicar igualdade e o resultado será 1 quando a comparação das saídas indicar diferença. Se o resultado for 1, ao menos 1 dos nodos v_a, v_b ou v_c está falho. Se o resultado for 0 e o nodo testador v_a também está sem-falha, então v_b e v_c também estão sem-falha. Mas se o nodo testador v_a estiver falho, o resultado dos testes não são confiáveis, portanto nenhuma conclusão pode ser tirada sobre o estado dos nodos v_b e v_c.

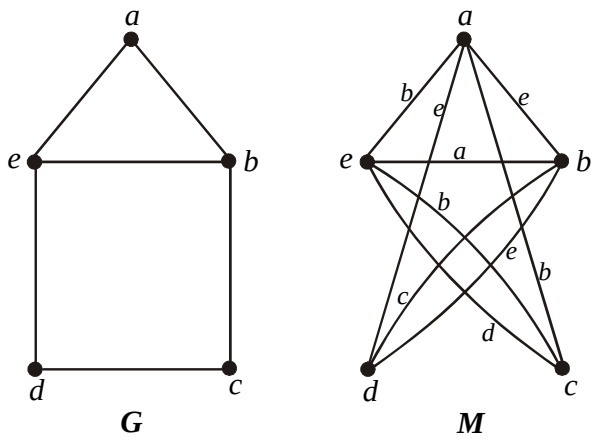

Figura 2.7: Um grafo *G* para um sistema de 5 nodos, e um multigrafo *M*.

Um *n*-hipercubo é um grafo $G=(V, E)$ onde V consiste de 2^n nodos, numerados de 00...0 (*n* bits) até 11...1 (*n* bits), e $\{v_i, v_j\} \in E$ se e somente se v_i e v_j têm somente um

bit de diferença. Desta forma todo nodo tem uma ligação com exatos *n* outros nodos. Se dois nodos v_i e v_j de um *n*-hipercubo possuem *d* bits diferentes, é dito que estes dois nodos têm *distância de Hamming* (*H*) igual a *d*. Então, em um *n*-hipercubo, ou simplesmente *n*-cubo, uma ligação existe entre v_i e v_j se e somente se $H(v_i \text{ e } v_j) = 1$. Como exemplo, um 3 hipercubo é mostrado através da figura 2.8(a) e um 4-hipercubo é mostrado na figura 2.8(b).

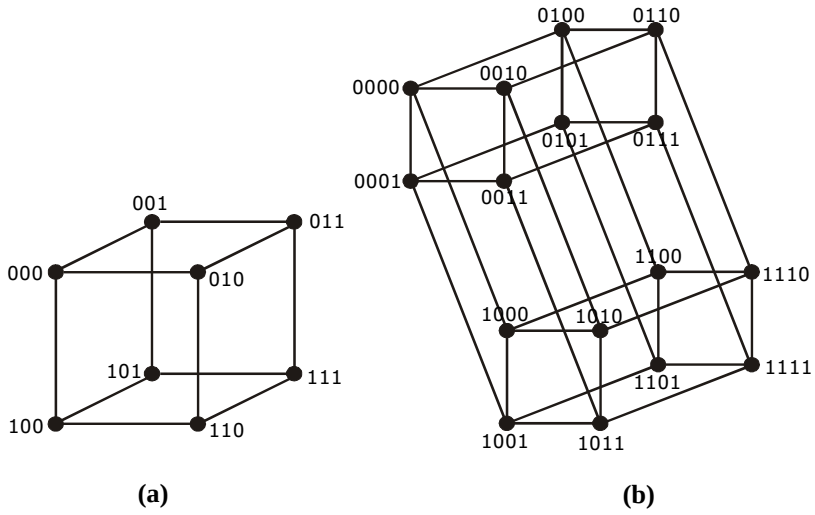

Figura 2.8: (a) Um 3-hipercubo, ou simplesmente um 3-cubo.
(b) Um hipercubo 4-dimensional, ou simplesmente um 4-cubo.

A diagnosticabilidade (*diagnosability*) para hipercubos é *n* para o diagnóstico baseado em comparações se $n \geq 5$ [35]. Conhecendo a *n*-diagnosticabilidade destes *n*-cubos sobre o modelo baseado em comparações, pode-se aplicar diretamente o algoritmo de diagnóstico proposto em [33].

Os hipercubos melhorados são obtidos pela adição de mais ligações, também conhecidos como desvios, sobre os hipercubos regulares. Estas ligações extras incrementam a habilidade de diagnóstico do sistema e alcançam um aperfeiçoamento

sobre muitas medidas [38] como o diâmetro e a densidade de trafego dos hipercubos melhorados. Estes hipercubos melhorados são denotados por (*n*, *k*)-cubo onde são adicionados 2^{n-1} desvios sobre as ligações já existentes em um *n*-cubo. Existe um desvio entre um par de nodos $b_n b_{n-1} ... b_{k+1} b_k b_{k-1} ... b_1$ e $b_n b_{n-1} ... b_{k+1} \bar{b}_k \bar{b}_{k-1} ... \bar{b}_1$, onde $\bar{b}_i$ é o complemento de b_i, e $k \in \{2, ..., n\}$ é a distancia de *Hamming* entre os pares de nodos ligados pelos desvios. O exemplo de um (3, 2)-cubo e um (3,3)-cubo são mostrados respectivamente através da figura 2.9(a) e 2.9(b), onde os desvios são mostrados através das linhas pontilhadas.

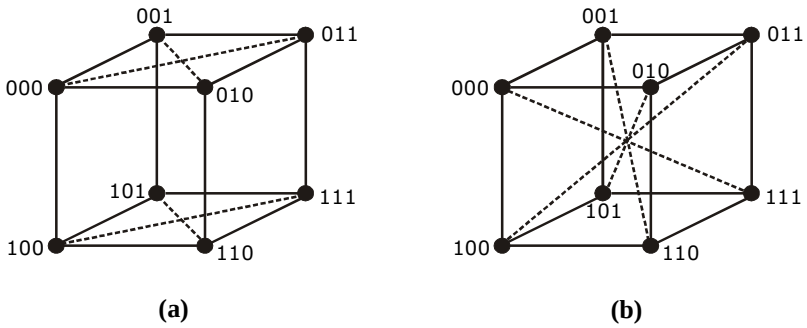

Figura 2.9: (a) Um (3,2)-cubo. (b) Um (3,3)-cubo. Os desvios são mostrados através das linhas pontilhadas.

A diagnosticabilidade dos hipercubos melhorados é incrementada para *n+1* sobre o diagnóstico baseado em comparações se $n \geq 6$. Conhecendo a (*n*+1)-diagnosticabilidade sobre o modelo baseado em comparações, pode-se aplicar o algoritmo de diagnóstico proposto em [33] para encontrar os nodos falhos de um *n*-cubo melhorado.

2.7 Diagnóstico Baseado em Comparações sobre Redes Borboletas

No modelo apresentado por Araki e Shibata [39] é estudada a diagnosticabilidade de redes borboletas [40] através do diagnóstico baseado em comparações. São propostos dois métodos de geração de testes em redes borboletas, um é chamado de comparação *one-way*, e o outro de comparação *two-way*.

O sistema é representado por um grafo $G=(V, E)$, onde cada nodo $u \in V$ representa um processador e cada aresta $(u, v) \in E$ representa um enlace de comunicação entre u e v. O diagnóstico baseado em comparações é realizado através do envio da mesma tarefa de entrada para um par $\{u, v\}$ de processadores e é realizada a comparação das respostas destas tarefas. A comparação é feita em um terceiro processador x que tem acesso a ambos os processadores u e v. O resultado da comparação é igual ou diferente. Assume-se que um processador tem acesso a outro se e somente se existe um enlace de comunicação entre eles. Então x é o processador que compara u e v se e somente se $(x, u) \in E$ e $(x, v) \in E$.

A comparação no sistema pode ser modelada por um multigrafo $M=(V, C)$. V é o conjunto de processadores. Uma aresta $\{u, v\}_x \in C$, onde x é o label da aresta (u, v), indica que x é o processador que compara de u e v. M é um multigrafo porque o mesmo par de processadores pode ser comparado por diferentes processadores. A figura 2.10 mostra um exemplo de um grafo de interconexão G e um multigrafo de comparação M. Por exemplo, nesta figura, através do multigrafo M, podemos notar que os nodos a e e

são comparados pelo nodo *b*, os nodos *a* e *b* são comparados pelo nodo *e*, e assim por diante.

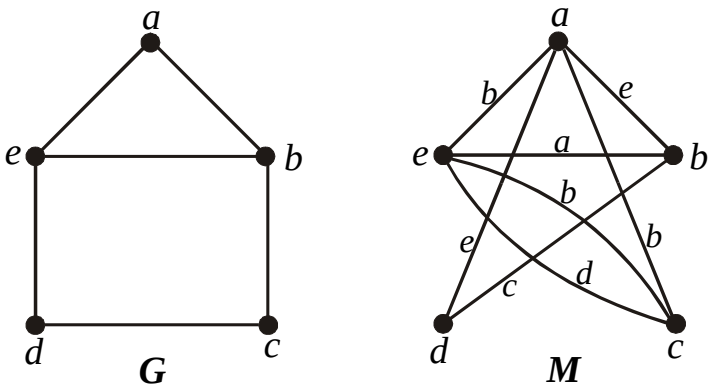

Figura 2.10: Um grafo de interconexão *G* e um multigrafo de comparação *M*.

Neste modelo, processadores falhos satisfazem às seguintes restrições: todas as falhas são permanentes; um processador falho produz saídas incorretas para suas entradas; a saída da comparação realizada por um processador falho não é confiável; e, dois processadores falhos com a mesma entrada não produzem a mesma saída.

O conjunto com o resultado de todas as comparações é chamado de síndrome do sistema. O objetivo é determinar todos os processadores falhos e sem-falhas do sistema através da análise desta síndrome. Um sistema é *t*-diagnosticável se, para toda síndrome, o número de processadores falhos não ultrapassa *t*.

Uma rede borboleta BF(k, r) é uma rede k-ária e r-dimensional e possui rk^r nodos. Cada nodo tem o label $\langle \ell; x_0x_1...x_{r-1} \rangle$, onde $0 \leq \ell \leq r-1$, $0 \leq x_i \leq k-1$, e $0 \leq i \leq r-1$. É chamado de ℓ o nível (*level*) dos nodos. Cada nodo $\langle \ell; x_0x_1...x_{r-1} \rangle$ é adjacente ao nodo

$\langle \ell+1; x_0 ... x_{\ell-1} y_\ell x_{\ell+1} ... x_{r-1} \rangle$ para $0 \le y_\ell \le k-1$, e ao nodo

$\langle \ell-1; x_0 ... x_{\ell-2} y_{\ell-1} x_\ell ... x_{r-1} \rangle$ para $0 \le y_{\ell-1} \le k-1$.

Como exemplos, a estrutura de uma BF(2, 3) e a estrutura de uma BF(3, 3) são mostradas nas figuras 2.11 e 2.12 respectivamente. Nestas figuras os nodos no nível 0 são replicados para facilitar a visualização.

No método chamado de comparação *one-way* baseado na estrutura de redes borboletas, cada nodo u no nível ℓ compara todo par de vizinhos no nível $\ell+1$. Por exemplo na figura 2.11, o nodo b realiza a comparação $\{a, d\}_b$ e o nodo c realiza a comparação $\{a, d\}_c$. Já na figura 2.12 o nodo b realiza três comparações $\{a, e\}_b$, $\{a, f\}_b$, $\{e, f\}_b$. No segundo método chamado de comparação *two-way*, cada nodo u no nível ℓ compara todo par de vizinhos no nível $\ell-1$ e no nível $\ell+1$. Na figura 2.11, nota-se que o nodo a realiza duas comparações $\{b, c\}_a$, {c, $f\}_a$. Na figura 2.12 o nodo a realiza seis comparações $\{b, c\}_a$, $\{b, d\}_a$, $\{c, d\}_a$, e $\{x, y\}_a$, $\{x, z\}_a$ e $\{y, z\}_a$. Cada nodo realiza $k(k-1)/2$ comparações no método comparação *one-way* e realizada $k(k-1)$ comparações no método comparação *two-way*.

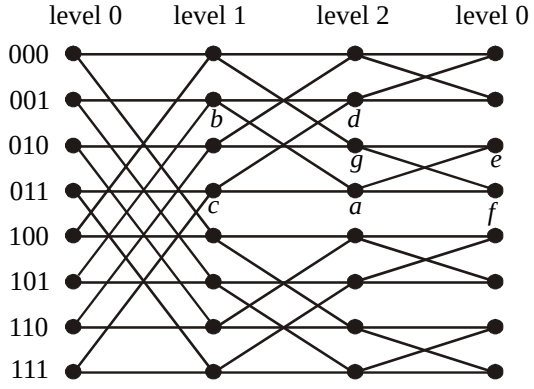

Figura 2.11: Estrutura de uma BF(2, 3) onde os nodos no nível 0 estão replicados.

A diagnosticabilidade de redes borboletas BF(k, r) é no máximo 2k. A diagnosticabilidade mostrada por este modelo para o método comparação *one-way* é k–2 e 2(k–2) no método comparação *two-way*. Um método melhorado de comparação também é proposto e aumenta a diagnosticabilidade para 2k, que é o maior valor possível. Araki e Shibata em [47] também mostram que os algoritmos de diagnóstico deste modelo são da ordem de tempo $O(k^2n)$.

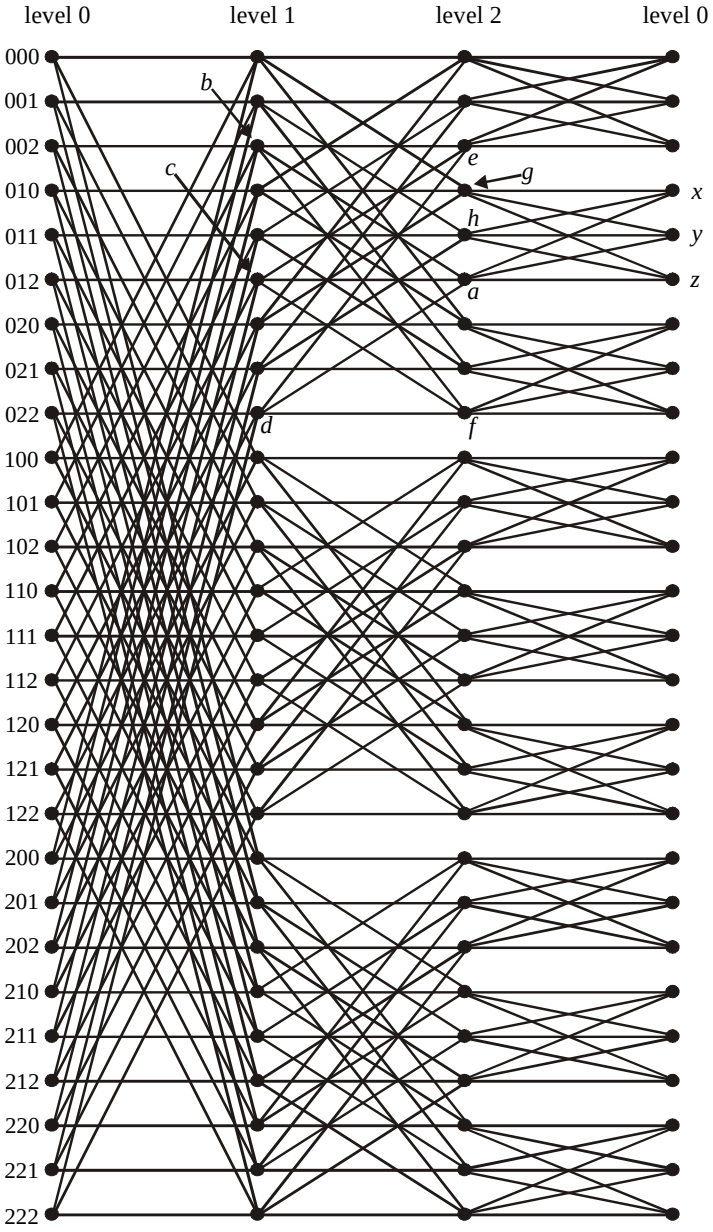

Figura 2.12: Estrutura de uma BF(3, 3) onde os nodos no nível 0 estão replicados.

2.8 Diagnóstico Baseado em Comparações Através de Cubos Cruzados

O modelo proposto por Fan [41] avalia a diagnosticabilidade de cubos cruzados considerando o diagnóstico baseado em comparações. Os cubos cruzados são uma importante variação dos hipercubos [48, 49, 50]. Ambos os cubos cruzados e hipercubos são grafos regulares que possuem o mesmo número de nodos, número de arestas e conectividade; ambos são recursivos por natureza. Mas o diâmetro dos cubos cruzados é aproximadamente a metade [41] do diâmetro dos hipercubos correspondentes. Um cubo cruzado n-dimensional contém uma árvore binária completa com 2^n–1 nodos e com todos os ciclos de tamanho de 4 a 2^n para $n \geq 2$, enquanto um hipercubo n-dimensional não possui estas propriedades [41].

Para a definição de um cubro cruzado, em [48, 49] é feita a seguinte definição: duas *strings* binárias $x = x_1x_0$ e $y = 1y_0$ são par-relacionadas (*pair-related*), denotado por $x \sim y$, se e somente se $(x, y) \in \{(00, 00), (10, 10), (01, 11), (11, 01)\}$; se x e y não são par-relacionados, é denotado $x \not\sim y$. Um cubo cruzado n-dimensional também chamado de CQ_n, é definido recursivamente assim como em [48, 49]. CQ_1 é o grafo completo com dois nodos com label 0 e 1, respectivamente. Para $n > 1$, CQ_n consiste de dois subcubos CQ^0_{n-1} e CQ^1_{n-1}. O nodo $u = 0u_{n-2} ...u_0$ do CQ^0_{n-1} e o nodo $v = 1v_{n-2} ...v_0$ do CQ^1_{n-1} são adjacentes se e somente se

1) $u_{n-2} = v_{n-2}$ se n é par, e
2) para $0 \leq i < \lfloor (n-1)/2 \rfloor$, $u_{2i+1}u_{2i} \sim v_{2i+1}v_{2i}$.

Como exemplo, a figura 2.13 mostra um cubo cruzado 3-dimensional CQ_3.

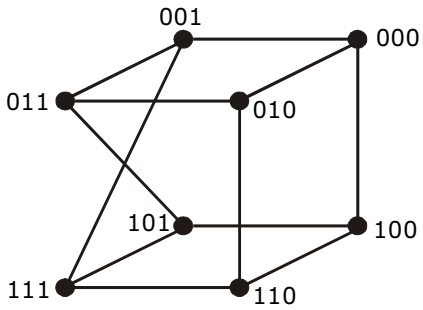

Figura 2.13: Um cubo cruzado 3-dimensional CQ_3.

O sistema é representado por um grafo $G=(V, E)$, onde todo nodo $v \in V$ representa um processador e cada aresta $(u, v) \in E$ representa um enlace de comunicação entre u e v. O sistema pode ser modelado como um multigrafo $M=(V, C)$ onde V representa o conjunto de processadores e C o conjunto de arestas. Toda aresta (v, w) de nome u é escrita como $(v, w)_u \in C$. Uma aresta $(v, w)_u \in C$ representa a comparação de v e w por u. M é um multigrafo porque o mesmo par de processadores pode ser comparado por diferentes processadores. Caso u esteja sem-falha, o resultado da comparação das saídas de v e w por u será 0 se v e w estiverem sem-falha. Este mesmo resultado será 1 se um dos dois processadores v ou w estiver falho. Mas caso u estiver falho, o resultado não é confiável.

O conjunto com o resultado de todas as comparações é chamado de síndrome do sistema. O objetivo é determinar todos os processadores falhos e sem-falhas do sistema através da análise desta síndrome. A diagnosticabilidade do sistema é definida em função de t, e para o sistema ser t-diagnosticável o número de processadores falhos não deve ultrapassar t.

Este modelo mostra que diagnosticabilidade para cubos cruzados sobre o diagnóstico baseado em comparação é n para $n \geq 4$. O algoritmo polinomial de diagnóstico apresentado em [33] pode ser usado para diagnosticar cubos cruzados n-dimensionais se o número de nodos falhos no cubo cruzado n-dimensional não ultrapassar n.

2.9 Modelo Genérico de Diagnóstico Distribuído Baseado em Comparações

No modelo genérico de diagnóstico distribuído e adaptativo em nível de sistema baseado em comparações [43], considera-se que um sistema S com N nodos é completamente conectado, e é também representado por um grafo $G=(V, E)$, onde V é um conjunto de vértices e E é um conjunto de arestas. Cada vértice do grafo corresponde a um nodo do sistema e as arestas correspondem aos enlaces de comunicação. Neste modelo os enlaces de comunicação não falham. Os nodos do sistema podem estar falhos ou sem-falhas e as mudanças de estado dos nodos são chamadas de eventos [29].

Este modelo usa a mesma estratégia hierárquica de testes implementada no algoritmo *Hi-ADSD with Timestamps* [29], mas não é limitado a falhas do tipo crash como os algoritmos baseados no modelo PMC. Um nodo sem-falha testa outro nodo do sistema para identificar seu estado. Um teste é realizado através do envio de uma tarefa para um par de nodos do sistema. Após o recebimento das duas saídas da execução destas tarefas, o testador compara estas saídas e, se a comparação indicar igualdade, os

nodos são considerados sem-falha. Por outro lado, se a comparação das saídas indicar diferença, pelo menos um dos nodos testados será considerado falho, mas sem concluir qual dos dois é o nodo falho.

As seguintes asserções são necessárias neste modelo:

1) a comparação realizada por um nodo sem-falha sobre as saídas produzidas por dois nodos sem-falha sempre resulta em igualdade;
2) a comparação realizada por um nodo sem-falha sobre as saídas produzidas por um nodo falho e qualquer outro nodo falho ou sem-falha sempre resulta em diferença;
3) o tempo para um nodo sem-falha produzir uma saída para uma tarefa é limitado.

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

Um multigrafo $M(S)$ [51], representa a maneira com que os testes são executados no sistema. $M(S)$ é um multigrafo direcionado definido sobre o grafo G, onde todos os nodos do sistema são sem-falha. Os vértices de $M(S)$ correspondem aos nodos do sistema S. Cada aresta de $M(S)$ representa que um nodo está enviando uma tarefa para ser executada por outro nodo, isto é, existe uma aresta do nodo i para o nodo j se o nodo i envia uma tarefa para ser executada pelo nodo j.

Considere como exemplo de um multigrafo $M(S)$ a figura 2.14, onde o nodo 0 envia um par de tarefas para os nodos 1 e 2, e também outro par de tarefas para os nodos 2 e 3. Para representar as diferentes arestas existentes entre um mesmo par de nodos,

cada aresta possui um identificador constituído do identificador do par de nodos que conecta esta tarefa, mais o identificador do outro nodo que também recebeu esta mesma tarefa e terá sua saída comparada com a saída desta tarefa. Neste exemplo as aresta do grafo são $(0, 1)_2$, $(0, 2)_1$, $(0, 2)_3$ e $(0, 3)_2$, todas direcionadas do nodo 0 para outros nodos do sistema. A aresta $(0, 1)_2$ indica que o nodo 0 está enviando uma tarefa para o nodo 1 e a saída desta tarefa será comparada com a saída da tarefa enviada do nodo 0 para o nodo 2, e obrigatoriamente deve existir a aresta $(0, 2)_1$.

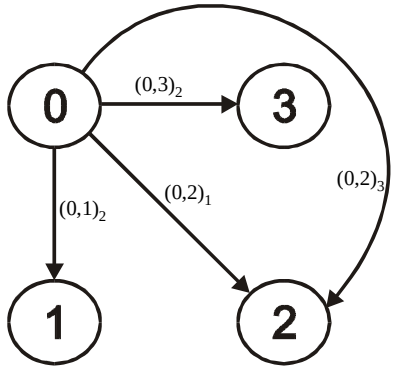

Figura 2.14: Exemplo de multigrafo *M*(*S*).

O trabalho [43] apresenta um algoritmo de diagnóstico hierárquico, adaptativo e distribuído em nível de sistema baseado em comparações *Hi-Comp* (*Hierarchical Comparison-Based Adaptive Distributed System-Level Diagnosis Algorithm*), que é especificado a seguir.

2.9.1 Especificação do Algoritmo *Hi-Comp*

No algoritmo *Hi-Comp*, um teste é realizado através do envio de uma tarefa para um par de nodos do sistema. Após o recebimento das duas saídas da execução destas

tarefas, o testador compara estas saídas e se a comparação indicar igualdade, os nodos são considerados sem-falha. Se a comparação das saídas indicar diferença, pelo menos um dos nodos testados será considerado falho, mas sem concluir qual dos dois é o nodo falho.

Neste algoritmo, um grafo $T(S)$ representa a estratégia de testes empregada. $T(S)$ é um hipercubo quando todos os nodos do sistema estão sem-falha. O grafo de testes de nodos sem-falha $T_i(S)$, é um grafo direcionado sobre $T(S)$ e mostra como a informação de diagnóstico é transmitida no sistema. Existe uma aresta em $T_i(S)$ do nodo *a* para o nodo *b* se existe uma aresta em $T(S)$ do nodo *a* para o nodo *b* e a distância de diagnóstico do nodo *i* para o nodo *a* for menor que a distância de diagnóstico do nodo *i* para o nodo *b*. A figura 2.15 mostra um grafo $T(S)$ para um sistema de 8 nodos.

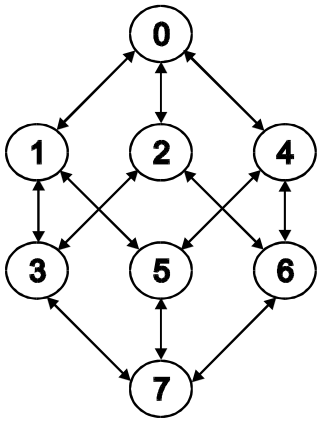

Figura 2.15: $T(S)$ para um sistema com 8 nodos.

Uma rodada de teste é um intervalo de tempo no qual todos os nodos sem-falha do sistema obtêm informação de diagnóstico sobre todos os nodos do sistema. Uma asserção é feita, na qual após o nodo *i* já ter testado o nodo *j* em uma certa rodada de teste, o nodo *j* não pode sofrer mais nenhum evento nesta rodada de teste.

A estratégia de testes agrupa os nodos em clusters assim como o algoritmo *Hi-ADSD with Timestamps* [29]. Cada cluster tem *N*/2 nodos. Uma função baseada na distância de diagnóstico define a lista de nodos sobre os quais o nodo *i* pode obter informação de diagnóstico sobre um nodo *p*.

Inicialmente um nodo *i* envia uma tarefa para seus nodos filhos em pares. Como exemplo, em um sistema de 16 nodos mostrado na figura 2.16, o nodo 0 envia uma tarefa para os nodos 1 e 2; em seguida envia outra tarefa para os nodos 4 e 8. Quando a quantidade de nodos filhos é impar, o último nodo filho é testado com seu antecessor. Por exemplo, no sistema com 8 nodos mostrado na figura 2.15, o nodo 0 envia uma tarefa para os nodos 1 e 2; em seguida envia outra tarefa para os nodos 2 e 4.

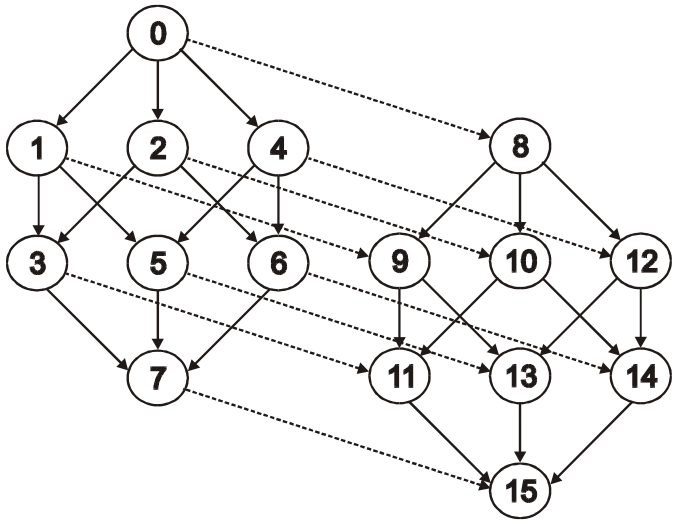

Figura 2.16: Um sistema com 16 nodos.

Quando o nodo *i* realiza o diagnóstico de dois nodos como sem-falha através da comparação do resultado das tarefas destes nodos, o nodo *i* obtém a informação de diagnóstico sobre o cluster correspondente a que cada um destes dois nodos pertencem.

Neste algoritmo é possível que o nodo *i* receba informações de diagnóstico do nodo *j* através de dois ou mais nodos *p* e *p'*, pois um nodo pode pertencer a mais de um cluster. Como exemplo, a figura 2.17 mostra um sistema com 8 nodos onde o nodo 7 pertence a três clusters diferentes em destaque. Por este motivo, o algoritmo implementa *timestamps* [29, 45] para determinar a ordem em que os eventos foram detectados e garantir que o nodo *i* receba sempre a informação de diagnóstico mais recente sobre o estado de todos os nodos do sistema. Quando o nodo *i* recebe informação de diagnóstico sobre o nodo *j* através do nodo *p*, o nodo *i* compara seu próprio *timestamp* sobre o nodo *j* com o *timestamp* de *p* sobre o nodo *j*, se a comparação indicar que o *timestamp* de *p* sobre o nodo *j* é mais recente, então o nodo *i* atualiza a sua informação de diagnóstico, caso contrário, o nodo *i* simplesmente desconsidera esta informação de diagnóstico recebida do nodo *p*.

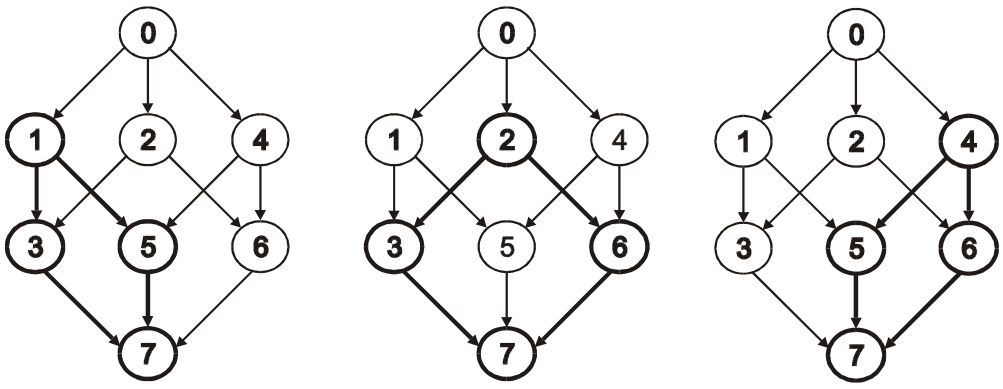

Figura 2.17: Uma divisão de clusters em um sistema de 8 nodos.

O algoritmo trabalha sobre três conjuntos: *U*, o conjunto dos nodos indefinidos; *F*, o conjunto dos nodos falhos; e, *FF*, o conjunto dos nodos sem-falha. Estes conjuntos possuem algumas propriedades: $U \cap F = \varnothing$, $U \cap FF = \varnothing$, $F \cap FF = \varnothing$ e

$U \cup F \cup FF = V$. Cada nodo do sistema sempre estará em um destes três conjuntos e ao final de cada rodada de testes o conjunto *U* sempre é vazio.

Quando o nodo *i* compara a saída de dois nodos *p* e *p*' e a comparação indicar igualdade, o nodo *i* identifica estes dois nodos como sem-falha. O nodo *i* coloca os nodos testados no conjunto *FF* removendo estes nodos do conjunto em que estes se encontravam. O nodo *i* também obtém informação de diagnóstico destes dois nodos sobre o cluster a que estes nodos pertencem. Cada cluster contém sempre $N/2$ nodos.

Quando um nodo *i* executa a comparação das saídas de dois nodos e a comparação indicar diferença, o nodo *i* classifica o estado destes dois nodos como *indefinido*, porque não é possível determinar se um ou ambos os nodos estão falhos. Assim, o nodo *i* coloca os dois nodos no conjunto *U* removendo estes nodos do conjunto em que se encontravam.

Se neste momento, o nodo *i* já tiver identificado algum nodo já classificado como sem-falha, o nodo *i* testa os dois nodos com este nodo sem-falha um de cada vez. Se a comparação das saídas indicar igualdade, então o nodo indefinido é considerado sem-falha, sendo mudado seu estado de indefinido para sem-falha, ou seja, o nodo é colocado no conjunto *FF*. Se a comparação indicar diferença, o nodo indefinido é considerado falho, sendo mudado seu estado de indefinido para falho, ou seja, o nodo é colocado no conjunto *F*. Mas se o nodo *i* não tiver identificado nenhum nodo sem-falha, os dois nodos indefinidos irão permanecer neste estado até que o nodo *i* chegue ao diagnóstico de algum nodo sem-falha que possa ser usado para fazer o diagnóstico dos nodos com estado indefinido.

Antes de o nodo *i* colocar um nodo *p* no conjunto *U*, o nodo *i* deve testar o nodo *p* com todos os nodos $k \in U$. Se todas as comparações indicarem diferença o nodo *i* coloca o nodo *p* no conjunto *U*. Mas se a comparação entre o nodo *p* e um nodo $k \in U$ indicar igualdade o testador classifica os nodos *p* e *k* como sem-falha e após todos os nodos $k \in U$ terem sido testados com o nodo *p* o testador classifica os outros nodos $\in U$ como falhos.

Quando um nodo $k \in U$ é identificado como sem-falha por um nodo *i*, o nodo *i* recebe informação de diagnóstico dos *N*/2 nodos do cluster do nodo testado. Se após o nodo *i* testar seus filhos, o conjunto *U* for vazio e o nodo *i* não tem informação de diagnóstico sobre algum nodo, o nodo *i* testa este nodo com qualquer nodo já identificado como sem-falha nesta rodada de testes.

Se após o nodo *i* testar seus filhos e o conjunto *FF* for vazio, isto é, todos os nodos estão classificados como indefinidos, o nodo *i* deve testar os nodos filhos dos seus filhos no grafo de testes, e assim por diante até que alguma comparação indique igualdade, ou até que o nodo *i* teste o nodo mais distante no grafo. O último nodo é testado com todos os nodos do sistema. Se qualquer comparação indicar igualdade, o testador classifica ambos os nodos como sem-falha, e o testador pode determinar o estado de todos os nodos do sistema através das informações de diagnóstico destes dois nodos, ou testando os nodos com estado indefinido com estes dois nodos sem-falha.

Se após terem sido testados todos os nodos o conjunto *FF* for vazio, o nodo *i* assume a si próprio como sem-falha e testa todos os nodos $k \in U$ consigo mesmo, um

por um. Neste momento, se a comparação indicar diferença o nodo *k* é considerado falho e se a comparação indicar igualdade o nodo *k* é considerado sem-falha. Desta forma, ao final da rodada de testes, todo nodo está no conjunto *F* ou *FF*, isto é, $F \cup FF = V$.

Capítulo 3

Um Algoritmo Hierárquico para Diagnóstico Distribuído Baseado em Comparações

Este capítulo apresenta o novo modelo genérico de diagnóstico distribuído e adaptativo baseado em comparações e o novo algoritmo *Hi-Dif*, baseado neste modelo. Também são apresentadas as provas da latência, do número de testes e da diagnosticabilidade do algoritmo *Hi-Dif*.

3.1 Um Novo Modelo Genérico de Diagnóstico Baseado em Comparações

O novo modelo genérico de diagnóstico distribuído e adaptativo em nível de sistema baseado em comparações, considera que um sistema S com N nodos é completamente conectado, e é também representado por um grafo $G=(V, E)$, onde V é

um conjunto de vértices e *E* é um conjunto de arestas. Cada vértice do grafo corresponde a um nodo do sistema e as arestas correspondem aos enlaces de comunicação. Neste modelo os enlaces de comunicação não falham. Os nodos do sistema podem estar falhos ou sem-falhas. Um nodo falho pode tanto estar com falha do tipo crash, como apenas estar respondendo de maneira incorreta à tarefa enviada como teste.

Este modelo usa a mesma estratégia hierárquica de testes implementada no algoritmo *Hi-ADSD with Timestamps* [29], mas não é limitado a falhas do tipo crash como os algoritmos baseados no modelo PMC. Um nodo sem-falha testa outro nodo do sistema para *classificar* seu estado. O modelo classifica os nodos do sistema em conjuntos de acordo com o resultado dos testes. Um teste é realizado através do envio de uma tarefa para um par de nodos do sistema. Após o recebimento das duas saídas da execução destas tarefas, o testador compara estas saídas e, se a comparação indicar igualdade, os nodos são classificados no mesmo conjunto de nodos. Por outro lado, se a comparação das saídas indicar diferença, os nodos são classificados em conjuntos distintos, de acordo com o resultado da tarefa. Um dos conjuntos contém os nodos sem-falha do sistema. Este modelo assume que o “usuário” do sistema é capaz de identificar o conjunto dos nodos sem-falha. O simples fato dos nodos serem classificados em mais de um conjunto indica que há nodos falhos no sistema.

Como este novo modelo faz uma classificação dos nodos em conjuntos a partir das saídas das tarefas recebidas e considerando a aplicação para o diagnóstico de sistemas com conteúdo replicado, este modelo permite: a identificação dos nodos que estão com falha do tipo crash; a identificação dos nodos sem-falha; e, a identificação

dos nodos falhos que possuem alteração de conteúdo. Além disso, permite ainda a identificação dos conjuntos de nodos que estão falhos e que possuem as mesmas modificações no conteúdo replicado.

As seguintes asserções são necessárias neste modelo:

1) a comparação realizada por um nodo sem-falha sobre as saídas produzidas por dois nodos sem-falha sempre resulta em igualdade;
2) a comparação realizada por um nodo sem-falha sobre as saídas produzidas por um nodo falho e outro sem-falha sempre resulta em diferença;
3) o tempo para um nodo sem-falha produzir uma saída para uma tarefa é limitado.

Uma diferença do modelo genérico de diagnóstico distribuído baseado em comparações, que foi descrito na seção 2.9, para o modelo proposto, é que aquele modelo possui a seguinte asserção: a comparação realizada por um nodo sem-falha sobre as saídas produzidas por um nodo falho e qualquer outro nodo falho ou sem-falha sempre resulta em diferença; ou seja, dois nodos falhos, ou um falho e outro sem-falha, não podem produzir as mesmas saídas para uma mesma tarefa de entrada. O modelo proposto elimina esta asserção, ou seja, se dois nodos quaisquer estiverem invadidos e forem vandalizados com as mesmas alterações e a comparação do resultado das tarefas dos dois nodos for realizada, esta pode resultar em igualdade.

Um multigrafo $M(S)$ [51], representa os testes executados no sistema. $M(S)$ é um multigrafo direcionado definido sobre o grafo G, onde todos os nodos do sistema são sem-falha. Os vértices de $M(S)$ correspondem aos nodos do sistema S. Cada aresta de

$M(S)$ representa que um nodo está enviando uma tarefa para ser executada por outro nodo, isto é, existe uma aresta do nodo *i* para o nodo *j* se o nodo *i* envia uma tarefa para ser executada pelo nodo *j*.

Neste modelo, um *evento* é definido como a mudança de estado de um nodo detectada pelo algoritmo de diagnóstico, ou seja, quando um nodo passa a responder de forma diferente à tarefa enviada como teste, em relação a forma como respondeu à tarefa pela última vez.

3.2 O Algoritmo *Hi-Dif*

Nesta seção é apresentado um novo algoritmo para o diagnóstico adaptativo distribuído e hierárquico baseado em comparações, chamado *Hi-Dif*, baseado no modelo de diagnóstico apresentado na seção anterior. Este novo algoritmo é voltado para o diagnóstico de alterações em nodos com conteúdo replicado. Como exemplo pode-se citar o diagnóstico de vandalismo em servidores Web. Este novo algoritmo ainda permite a identificação dos nodos que possuem um mesmo tipo de modificação de conteúdo e diferenciá-los de outros nodos que possuem outros tipos de modificação. Por fim, é descrito como é tratado o diagnóstico dinâmico de eventos e qual o procedimento de recuperação de falhas utilizado pelo algoritmo.

3.2.1 Descrição do Algoritmo *Hi-Dif*

O novo algoritmo, *Hi-Dif*, é dito distribuído porque o diagnóstico é executado por todos os nodos do sistema, adaptativo porque os próximos testes de um nodo são

baseados nos resultados de testes anteriores, e é dito hierárquico porque na estratégia de testes os nodos são divididos em clusters. Este algoritmo é baseado no modelo apresentado na subseção 3.1.

O multigrafo [33, 51, 52] de testes do sistema, $M(S)$, representa a maneira com que os testes são realizados. $M(S)$ é um multigrafo direcionado definido sobre o grafo G, onde todos os nodos do sistema são sem-falha. Os vértices de $M(S)$ correspondem aos nodos do sistema S. Cada aresta de $M(S)$ representa que um nodo está enviando uma tarefa para ser executada por outro nodo, isto é, existe uma aresta do nodo i para o nodo j se o nodo i envia uma tarefa para ser executada pelo nodo j.

Para representar as diferentes arestas existentes entre um mesmo par de nodos, cada aresta possui um identificador constituído do identificador do par de nodos que conecta esta tarefa, mais o identificador do outro nodo que também recebeu esta mesma tarefa e terá sua saída comparada. Desta forma se o nodo a envia uma tarefa para os nodos b e c, existe uma aresta no multigrafo $M(S)$ identificada por $(a, b)_c$ e existe outra tarefa em $M(S)$ identificada por $(a, c)_b$. Assim, sempre que existir uma aresta $(a, b)_c$ em $M(S)$, deve, obrigatoriamente, existir outra aresta $(a, c)_b$ em $M(S)$.

Considere como exemplo de um multigrafo $M(S)$ a figura 3.1, onde o nodo 0 envia um par de tarefas para si próprio e para o nodo 1, e também outro par de tarefas para si próprio e para o nodo 2. Neste exemplo as aresta do grafo são $(0, 0)_1$, $(0, 1)_0$, $(0, 0)_2$ e $(0, 2)_0$, todas direcionadas do nodo 0 para outros nodos ou para si próprio. A aresta $(0, 1)_0$ indica que o nodo 0 está enviando uma tarefa para o nodo 1 e a saída desta

tarefa será comparada com a saída da tarefa enviada do nodo 0 para o próprio nodo 0, e obrigatoriamente deve existir a aresta $(0, 0)_1$.

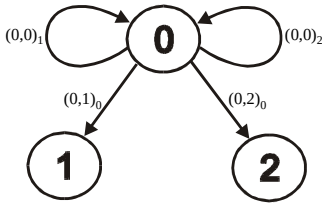

Figura 3.1: Exemplo de multigrafo *M*(*S*).

O grafo de testes de nodos sem-falha, *T*(*S*), é um grafo direcionado do sistema, definido sobre o multigrafo *M*(*S*). Este grafo representa a estratégia de testes empregada. *T*(*S*) é um hipercubo quando todos os nodos do sistema estão sem-falha e quando a quantidade de nodos do sistema é uma potência de 2. A figura 3.2 mostra um grafo *T*(*S*) para um sistema de 8 nodos.

A *distância de diagnóstico* entre o nodo *i* e o nodo *j* é definida como a menor distância entre o nodo *i* e o nodo *j* em *T*(*S*), isto é, o caminho com o menor número de arestas entre o nodo *i* e o nodo *j* no grafo *T*(*S*). Apesar de ser o menor caminho, pode existir mais de um caminho entre estes dois nodos com a mesma distância de diagnóstico. Por exemplo, na figura 3.2, a distância de diagnóstico entre o nodo 0 e o nodo 3 é 2, pois, apesar de existirem dois caminhos, o menor caminho entre esses nodos possui duas arestas.

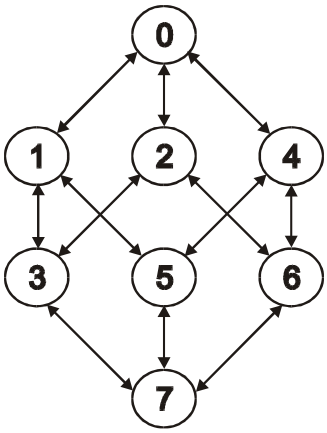

Figura 3.2: $T(S)$ para um sistema com 8 nodos.

O grafo de testes de nodos sem-falha do nodo i, $T_i(S)$, é um grafo direcionado sobre $T(S)$ e mostra como a informação de diagnóstico é transmitida no sistema. Existe uma aresta em $T_i(S)$ do nodo a para o nodo b se existe uma aresta em $T(S)$ do nodo a para o nodo b e a distância de diagnóstico do nodo i para o nodo a for menor que a distância de diagnóstico do nodo i para o nodo b. A figura 3.3 mostra a $T_0(S)$ para um sistema de 8 nodos. Chamamos de *filhos* do nodo i na $T_i(S)$, os nodos que possuem ligação direta com o nodo i. No exemplo da figura 3.3 os filhos do nodo 0 são os nodos 1, 2 e 4.

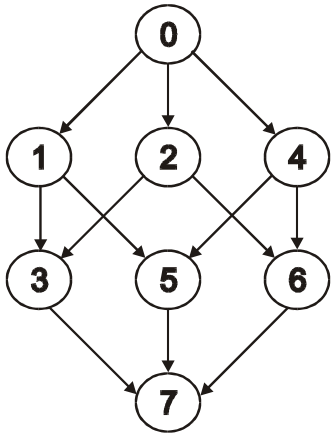

Figura 3.3: $T_0(S)$ para um sistema com 8 nodos.

Uma *rodada de teste* é um intervalo de tempo onde todos os nodos sem-falha do sistema obtêm informação de diagnóstico sobre todos os nodos do sistema. Uma asserção é feita, na qual após o nodo *i* já ter testado o nodo *j* em uma certa rodada de teste, o nodo *j* não pode sofrer mais nenhum evento nesta rodada de testes. Esta asserção é necessária para garantir a propagação das informações de diagnóstico pelo sistema [29].

A estratégia de testes agrupa os nodos em clusters como o algoritmo *Hi-ADSD with Timestamps* [29]. Uma função $c_{i,s,p}$ baseada na distância de diagnóstico define a lista de nodos sobre os quais o nodo *i* pode obter informação de diagnóstico através e a partir de um nodo *p*, com distância de diagnóstico menor ou igual a s –1 na $T_i(S)$. Neste algoritmo s é sempre igual a $log_2 N$, o que implica que cada cluster tem sempre $N/2$ nodos. A figura 3.4 mostra esta divisão de clusters para um sistema de 8 nodos para $T_0(S)$. Os clusters são: (a) $c_{0,3,1}$: nodos {1, 3, 5, 7}, (b) $c_{0,3,2}$: nodos {2, 3, 6, 7} e (c) $c_{0,3,4}$: nodos {4, 5, 6, 7}.

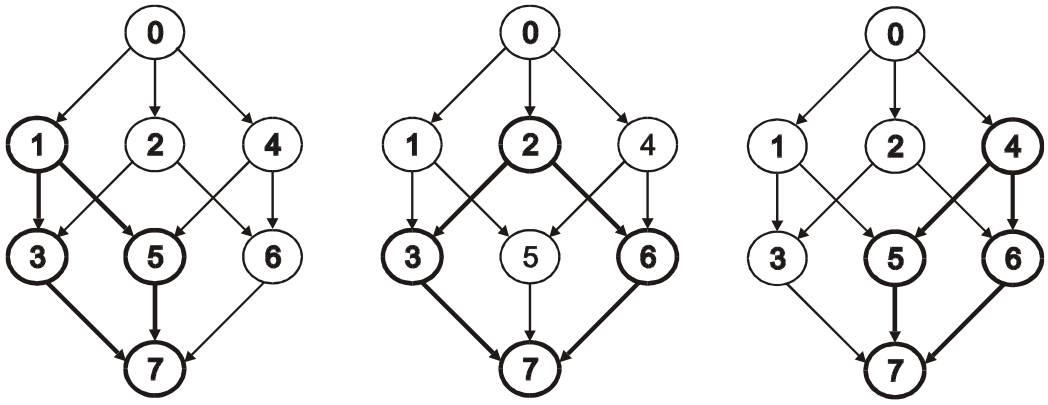

Figura 3.4: Divisão de clusters para um sistema de 8 nodos para $T_0(S)$.

Em comparação com outros modelos de diagnóstico, o modelo do algoritmo *Hi-Dif* tem uma diferença importante: a asserção em que a comparação realizada por um nodo sem-falha sobre as saídas produzidas por um nodo falho e outro nodo falho sempre resulta em diferença foi eliminada. Para eliminar esta asserção, neste algoritmo, um nodo *i* sempre se considera como sem-falha. Além disso, a estratégia de testes dos nodos será a de enviar tarefas em pares, sempre uma para si próprio e outra para um nodo a ser testado. Ou seja, um nodo *i* testador, sempre envia uma tarefa para si próprio e para outro nodo *j* e compara o resultado destas tarefas.

Se a comparação do resultado destas tarefas do próprio nodo *i* e do nodo *j* indicar igualdade, o nodo *i* irá obter informação de diagnóstico deste nodo *j*. Desta forma o nodo *i* sempre considera como sem falha somente os nodos que gerarem saída para a tarefa enviada, igual à saída gerada pela tarefa enviada ao próprio nodo testador.

Por este motivo, este algoritmo possui as seguintes asserções sobre o observador externo que é o “usuário” do algoritmo: este observador externo é quem realiza o diagnóstico do sistema e este observador externo é confiável e não pode falhar em hipótese alguma. Além disso, o observador externo sempre identifica corretamente se um nodo está correto a fim de se obter os dados de diagnóstico do sistema. Desta forma, este observador nunca irá escolher um nodo que esteja falho – ou no caso especifico a que se aplica o algoritmo, um nodo invadido ou com conteúdo incorreto – para obter as informações de diagnóstico do sistema.

Desta forma, no algoritmo *Hi-Dif* os testes são realizados através do envio da tarefa para 2 nodos do sistema que a executam e devolvem os resultados para o nodo

testador. Além disso, no algoritmo um destes nodos sempre será o próprio nodo testador. Assim, quando um nodo testador *i* envia uma tarefa para o par de nodos *i* e *x*, o nodo *i* considera o nodo *x* como sem-falha quando o resultado das tarefas enviadas aos nodos *i* e *x* indicar igualdade. Caso contrário, o nodo *x* será considerado falho.

3.2.2 Execução do Algoritmo

Inicialmente o nodo testador *i* envia uma tarefa para si próprio e outra para seu primeiro filho. Após receber os resultados desta tarefa, o nodo *i* envia outra tarefa para si próprio e para seu próximo filho, e assim por diante até que sejam enviadas tarefas para todos os seus filhos. Por exemplo, em um sistema de 8 nodos, quando os filhos do nodo 0 estão sem-falha, o nodo 0 envia a primeira tarefa para os nodos 0 e 1, ele próprio e seu primeiro filho; depois ele envia outra tarefa para os nodos 0 e 2, ele próprio e seu próximo filho, e por fim ele envia outra tarefa para os nodos 0 e 4, que é seu último filho.

Quando o nodo *i* diagnosticar que um de seus filhos está sem-falha, ou seja, o resultado da comparação das tarefas enviadas ao seu filho e ao nodo *i* resultar em igualdade, ele obtém informação de diagnóstico do cluster ao qual este nodo sem-falha pertence. Por exemplo, na figura 3.4, quando o nodo 0 diagnosticar o nodo 1 como sem-falha, ele obtém através do nodo 1 informação de diagnóstico sobre todo o cluster do nodo 1, ou seja, dos nodos 3, 5 e 7. Da mesma forma quando o nodo 0 diagnosticar o nodo 2 como sem-falha, ele obtém através do nodo 2 informação de diagnóstico sobre todo o cluster do nodo 2, ou seja, dos nodos 3, 6 e 7.

No algoritmo *Hi-Dif* é possível que o nodo *i* receba informações de diagnóstico do nodo *j* através de dois ou mais nodos *p* e *p'*, porque um nodo pode pertencer a mais de um cluster como mostrado na figura 3.4, onde o nodo 0 pode obter informação de diagnóstico do nodo 3 e 7 através dos nodos 1 e 2. Por este motivo, é necessário garantir que o nodo *i* sempre obtenha a informação mais recente sobre os outros nodos do sistema. Para garantir isso, o algoritmo implementa *timestamps* [29, 45]. Os *timestamps* são implementados através de contadores de troca de estados, que cada nodo possui para todos os nodos do sistema. Essa estratégia permite determinar a ordem em que os eventos foram detectados e garantir que o nodo *i* receba sempre a informação de diagnóstico mais recente sobre o estado de todos os nodos do sistema.

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

Quando o nodo *i* executar a comparação entre as saídas de um par de tarefas, enviadas uma ao próprio nodo *i* e outra ao um outro nodo *j*, e estas não indicarem igualdade, o nodo *i* classifica o nodo testado *j* como falho e não recebe informações de diagnóstico do cluster deste nodo *j*.

Se após o nodo *i* testar todos os seus filhos em uma rodada de testes ainda existirem nodos do sistema sobre os quais ainda não se obteve informações, o nodo *i* começa a testar diretamente os nodos, um de cada vez. Neste caso o nodo *i* pega o

próximo nodo como menor identificador que ainda não obteve informação sobre seu estado, e que a distância de diagnóstico seja igual a 2, ou seja, um nodo que seja filho de um dos seus filhos. Após já ter testado os filhos dos seus filhos e ainda restarem nodos em que não se tem informação sobre seus estados, o nodo *i* testa os nodos com distância de diagnóstico igual a 3 e assim por diante até que se teste o último nodo que não se tenha informação sobre seu estado, que está a uma distância $log_2 N$.

Como exemplo, a figura 3.5 mostra um sistema de 8 nodos onde os nodos 2 e 4 são falhos. Neste exemplo, o nodo 0 testa seus filhos, isto é, testa o nodo 1 e descobre que está sem-falha e recebe informações de diagnóstico sobre os nodos 3, 5 e 7, e em seguida testa os nodos 2 e 4 e descobre que estão falhos e não recebe nenhuma informação de diagnóstico. Neste momento o nodo 0 já testou todos os seus filhos ainda não tem nenhuma informação sobre o estado do nodo 6, desta forma o nodo 0 realiza mais um teste enviando um par de tarefas para o nodo 0 e para o nodo 6.

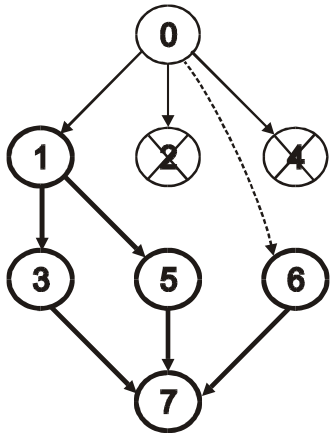

Figura 3.5: Sistema de 8 nodos onde os nodos 2 e 4 são falhos.

Mesmo após o nodo *i* ter testado todos os seus filhos e restarem *x* nodos do sistema dos quais não se conhece seu estado, isto não indica necessariamente que o nodo *i* irá realizar *x* testes extras para completar o diagnóstico do sistema. Isto se deve porque após o nodo *i* testar o primeiro destes *x* nodos, se este nodo estiver sem-falha, ele irá obter informação de diagnóstico sobre o restante dos nodos do cluster do nodo testado.

Como exemplo, a figura 3.6 mostra um sistema de 8 nodos onde os nodos 1, 2 e 4 estão falhos. Neste exemplo, o nodo 0 testa seus filhos, isto é, testa os nodos 1, 2 e 4 e descobre que estão falhos e não recebe nenhuma informação de diagnóstico sobre os demais nodos do sistema. Neste momento o nodo 0 não tem informações de diagnóstico sobre os nodos 3, 5, 6 e 7, e então testa o primeiro nodo com distância de diagnóstico igual a 2 e com menor identificador, ou seja, testa o nodo 3. Como o nodo 3 está sem-falha, o nodo 0 obtém informações sobre o nodo 7. O nodo 0 não obtém informação diagnóstico do nodo 5 através do nodo 3, pois a distância de diagnóstico do nodo 0 até o nodo 3 somada à distância de diagnóstico do nodo 3 até o nodo 5 é maior que a distância de diagnóstico do nodo 0 até o nodo 5. Desta forma após o nodo 0 ter testado o nodo 3, ele testa os nodos 5 e 6 que também irão repassar informações de diagnóstico sobre o nodo 7. Após o nodo 0 completar os testes dos filhos dos seus filhos, ou seja dos nodos 3, 5 e 6, ele não precisará testar o nodo 7 pois já recebeu informação sobre este através dos nodos 3, 5 e 6.

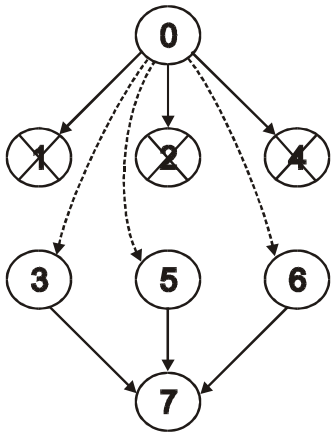

Figura 3.6: Sistema de 8 nodos com os nodos 1, 2 e 4 falhos.

3.2.3 Classificação de Nodos com Mesmo Resultado

Além de apresentar o diagnóstico do sistema, o novo algoritmo também executa uma classificação dos nodos do sistema em conjuntos. Estes conjuntos são construídos de forma a permitir a fácil identificação de quais nodos do sistema estão respondendo com o mesmo resultado para as tarefas enviadas como teste. Um dos objetivos desta classificação é, por exemplo, permitir a fácil identificação de quais nodos estão com falha do tipo crash, ou ainda permitir identificar quais conjuntos de nodos estão invadidos e vandalizados com o mesmo tipo de alteração sobre o conteúdo que deveria estar replicado.

Para isso, o algoritmo classifica os nodos em conjuntos com identificador numérico e sequencial e estes conjuntos são utilizados para a seguinte classificação: nodos com falhas do tipo crash; nodos que possuem o conteúdo correto, ou seja, que estão respondendo corretamente às tarefas; e, nodos com falhas por estarem respondendo às tarefas enviadas de forma diferente da considerada correta.

Como a identificação dos conjuntos é numérica e sequencial, foi padronizado o conjunto com identificador 0 para os nodos que possuem falha do tipo crash; o conjunto com identificador 1 para os nodos que não estão falhos, ou seja, nodos que estão com os dados replicado inalterados ou corretos; o conjunto com identificador 2 é utilizado para nodos que respondem da mesma maneira, mas incorretamente para as tarefas enviadas como teste; o conjunto com o identificador 3 para nodos que também respondem da mesma maneira, mas incorretamente e com resposta diferente da resposta dos nodos que estão no conjunto 2; e assim por diante. Desta forma, podemos identificar quais nodos já estão invadidos e vandalizados, caso exista algum nodo nesta situação, e permitir diferenciá-los, por exemplo, dos nodos que estão com falhas do tipo crash.

Para isso, sempre que um nodo testador envia uma tarefa para um par de nodos, por exemplo, um nodo *y* e o próprio nodo testador *i*, além de realizar a comparação do resultado das tarefas e classificar o nodo *y* como falho ou sem-falha de acordo com o resultado da comparação, o testador coloca o nodo *y* no conjunto 0 se este nodo não responder ao teste, coloca o nodo testado no conjunto 1 se o resultado da tarefa enviado ao nodo for igual ao resultado da tarefa enviada ao próprio nodo testador. Mas se o resultado da tarefa nodo *y* for diferente do resultado da tarefa do próprio nodo *i*, o testador começa a comparar o resultado da tarefa do nodo *y* com o resultado de um dos nodos de cada um dos conjuntos começando pelo conjunto de identificador 2. Desta forma o algoritmo compara o resultado do nodo *y* com o resultado de qualquer nodo do conjunto 2, se o resultado for o mesmo, o nodo *y* será colocado neste conjunto, caso contrário o algoritmo continua a busca, comparando o resultado da tarefa enviada ao nodo *y* com o resultado de qualquer nodo do conjunto 3, se o resultado for o mesmo, o nodo *y* será colocado neste conjunto, e assim por diante.

Desta forma ao ser encontrado um conjunto em que exista algum nodo que possui resultado igual ao resultado da tarefa enviada para o nodo *y*, o nodo *y* será colocado neste conjunto e o algoritmo não continua a busca. Caso sejam consultados todos os conjuntos a partir do conjunto 2, e mesmo assim não foi encontrado um conjunto que possui um nodo que tenha resultado igual ao do nodo *y*, será criado um novo conjunto com título indicando o número do maior conjunto já criado, somando-se uma unidade. Por exemplo: se o maior conjunto já criado for o conjunto 3, e um nodo do conjunto 2 não possuir o mesmo resultado do nodo *y*, e também um nodo do conjunto 3 não possuir o mesmo resultado do nodo *y*, será criado o conjunto com identificador 4 e colocado o nodo *y* neste conjunto.

3.2.4 Especificação do Algoritmo

O algoritmo *Hi-Dif* agrupa os nodos em dois conjuntos de estados: o conjunto dos nodos falhos: *F*, e o conjunto dos nodos sem-falha: *FF*. Estes conjuntos são sempre disjuntos e a união deles sempre resulta em *V*, isto é $F \cap FF = \emptyset$ e $F \cup FF = V$. Além disso, o algoritmo utiliza também uma lista de conjuntos, *result-set-list*, para classificar o nodo de acordo com o resultado da tarefa enviada. Cada nodo do sistema mantém estes 2 conjuntos *F* e *FF* e mantém também a lista de conjuntos *result-set-list*. Ao final de cada rodada de testes cada nodo deve estar em um dos dois conjuntos *F* e *FF* e deve estar em exatamente um dos conjuntos da lista *result-set-list*.

Quando o nodo *i* compara as saídas de uma tarefa realizada pelo próprio nodo *i* e por outro nodo *p*, e essa comparação indicar igualdade, o nodo *i* identifica o nodo *p*

como sem-falha. O nodo *i* coloca o nodo *p* no conjunto *FF* retirando-o do conjunto *F* caso estivesse neste conjunto. Além disso, o nodo *p* é colocado no conjunto 1 da lista *result-set-list*, ou seja, é colocado no conjunto dos nodos com conteúdo correto. Em outras palavras:

```
envia_tarefa( i, p );
SE ( resultado(i) == resultado(p) )
ENTÃO
        F = F – {p};
        FF = FF + {p};
        result-set-list[conjunto 1] = result-set-list[conjunto 1] + {p};
FIM_SE
```

Quando o nodo *i* identifica um nodo sem-falha, ele obtém desse nodo informações de diagnóstico sobre todo o cluster desse nodo. Além disso, como são utilizados *timestamps* para datar as informações, o nodo *i* deve testar se as informações que está recebendo são mais novas que as que ele já possui. Caso as informações que está recebendo sejam mais recentes, o nodo *i* deve substituir suas informações pelas que está recebendo, caso contrário deve simplesmente descartar estas informações. Em outras palavras:

```
envia_tarefa( i, p );
SE ( resultado(i) == resultado(p) )
ENTÃO
        F = F – {p};
        FF = FF + {p};
        result-set-list[conjunto 1] = result-set-list[conjunto 1] + {p};

        OBTER informações de diagnóstico do cluster de p;
                COMPARAR timestamps dos nodos do cluster;
                ATUALIZAR informações locais se necessário;
FIM_SE
```

Quando o nodo *i* envia uma tarefa para ser realizada pelo próprio nodo *i* e por outro nodo *p*, mas o nodo *p* não responde, o nodo *i* identifica o nodo *p* como falho. O nodo *i* coloca o nodo *p* no conjunto *F* retirando-o do conjunto *FF* caso estivesse neste conjunto. Além disso, o nodo *p* é colocado no conjunto 0, ou seja, é colocado no conjunto dos nodos com falha do tipo crash da lista *result-set-list*. Em outras palavras:

```
envia_tarefa( i, p );
SE ( resultado(p) == ∅ )
ENTÃO
        FF = FF – {p};
        F = F + {p};
        result-set-list[conjunto 0] = result-set-list[conjunto 0] + {p};
FIM_SE
```

Quando o nodo *i* compara as saídas de uma tarefa realizadas pelo próprio nodo *i* e por outro nodo *p*, e essa comparação indicar diferença, o nodo *i* identifica o nodo *p* como falho. O nodo *i* coloca o nodo *p* no conjunto *F* retirando-o do conjunto *FF* caso estivesse neste conjunto. Em outras palavras:

```
envia_tarefa( i, p );
SE ( resultado(i) != resultado(p) )
ENTÃO
        FF = FF – {p};
        F = F + {p};
FIM_SE
```

Além disso, quando o nodo *i* identifica um nodo *p* como falho porque o resultado da tarefa do nodo *p* foi diferente do resultado da tarefa enviada ao nodo *i*, é procurado um conjunto na lista *result-set-list* para se colocar o nodo *p*. Essa procura começa pelo conjunto 2 da lista *result-set-list*. Para cada conjunto a partir do conjunto 2, é verificado se o primeiro nodo *x* deste conjunto possui resultado para a tarefa enviada a este nodo *x* igual ao resultado da tarefa do nodo *p*. Caso esta verificação seja confirmada, o nodo *p* será colocado no mesmo conjunto do nodo *n* na lista *result-set-list*. Caso contrário, será criado um novo conjunto na lista *result-set-list* e colocado o nodo *p* neste conjunto. Em outras palavras:

```
envia_tarefa( i, p );
SE ( resultado(i) != resultado(p) )
ENTÃO
        FF = FF – {p};
        F = F + {p};

        id_conj = NULO;
        PARA (x começando em 2)
        ENQUANTO ( x <= result-set-list.id_maior_conjunto E id_conj == NULO )
        FAÇA
                SE ( resultado(um nodo de result-set-list[conjunto x]) == resultado(p) )
                ENTÃO
                        id_conj = x;
                FIM_SE
        FIM_PARA

        SE ( id_conj != NULO )
        ENTÃO
                result-set-list[conjunto id_conj] =
                                                result-set-list[conjunto id_conj] + {p};
        SENÃO
                id_novo_conj = result-set-list.criar_novo_conjunto;
                result-set-list[conjunto id_novo_conj] = {p};
        FIM_SE
FIM_SE
```

Desta forma, o nodo *i* testa primeiramente seus filhos na $T_i(S)$, enviando uma tarefa sempre para o próprio nodo *i* e para o seu próximo filho a testar. Assim, através da comparação do resultado das tarefas enviadas, o nodo *i* determina o estado de todos os seus nodos filhos.

Quando o nodo *i* acaba de testar seus filhos, e ainda restam nodos não testados em que não se tem informação sobre seu estado, o nodo *i* testa diretamente estes nodos, um por um. O nodo *i* então testa primeiramente os nodos com menor identificador e com distância de diagnóstico igual a 2, ou seja, os nodos filhos dos seus filhos. Se depois de testados estes nodos com distância de diagnóstico igual a 2, ainda existirem nodos que não foram testados e que não se obteve informação sobre seu estado na rodada de testes corrente, o nodo *i* então testa os nodos, por ordem de identificador, com distância de diagnóstico igual a 3, e assim por diante até que se conheça o estado de todos os nodos do sistema. A cada teste, se o nodo *i* identificar um nodo sem-falha, será solicitada pelo nodo *i* informação de diagnóstico necessária sobre os nodos do cluster do nodo testado.

Neste algoritmo nota-se claramente a necessidade do nodo *i* testar primeiramente seus filhos, em seguida testar os nodos com distância de diagnóstico igual a 2 e que ainda não se obteve informações sobre seus estados, e assim por diante. Para facilitar esta tarefa, a estratégia utilizada pelo algoritmo foi criar uma lista com todos os nodos do sistema, mas ordenada por distância de diagnóstico do nodo *i* para cada nodo, e dentro desta ordem de distância de diagnóstico, ordenar também pelo identificador dos nodos. Por exemplo, considerando a $T_0(S)$ mostrada na figura 3.3 para um sistema de 8 nodos, a lista do nodo 0 de nodos a serem testados seria: 1, 2, 4, 3, 5, 6 e 7. Em outras palavras:

```
TO_TEST = {ALL NODES};
TO_TEST = Ordenar_por_Distância_Diagnóstico_e_Identificador( TO_TEST );

REPETIR
        p = próximo_nodo em TO_TEST;
        TO_TEST = TO_TEST – {p}

        envia_tarefa( i, p );

        ‘comparar as saídas das tarefas de i e p,
         para identificar o estado do nodo p,
         identificar o conjunto de p na lista result-set-list, e
         obter informação de diagnóstico do cluster
         de p caso p esteja sem-falha.’

ATÉ ( foi encontrado o estado de todos os nodos )
```

Assim, ao final de cada rodada de testes, todo nodo *i* sem-falha terá todos os nodos do sistema em um dos dois conjuntos *F* ou *FF*, isto é $F \cup FF = V$. Além disso todos os nodos do sistema também estarão em um dos conjuntos da lista *result-set-list*. Um nodo que estiver no conjunto 0 da lista *result-set-list*, necessariamente também estará no conjunto *F*. Um nodo que estiver no conjunto 1 da lista *result-set-list*, necessariamente estará no conjunto *FF*. E um nodo que estiver em qualquer outro conjunto da lista *result-set-list*, necessariamente também estará no conjunto *F*.

Abaixo é apresentado o algoritmo em pseudo-código.

```
Algoritmo rodando no nodo i:

F = EMPTY; FF = EMPTY;

REPETIR PARA SEMPRE

       Inicializar( result-set-list );
       TO_TEST = {ALL NODES};
       TO_TEST = Ordenar_por_Distância_Diagnóstico_e_Identificador( TO_TEST );

       REPETIR
              p = próximo_nodo em TO_TEST;

              envia_tarefa( i, p );
              TO_TEST = TO_TEST – {p}

              SE ( resultado(p) == ∅ )
              ENTÃO
                     FF = FF – {p};
                     F = F + {p};
                     result-set-list[conjunto 0] =
                                          result-set-list[conjunto 0] + {p};

              SENÃO SE ( resultado(i) == resultado(p) )
              ENTÃO
                     F = F – {p};
                     FF = FF + {p};
                     result-set-list[conjunto 1] =
                                          result-set-list[conjunto 1] + {p};

                     OBTER informações de diagnóstico do cluster de p;
                            COMPARAR timestamps dos nodos do cluster;
                            ATUALIZAR informações locais se necessário;
                            TO_TEST = TO_TEST – {nodos com info atualizadas};

              SENÃO SE ( resultado(i) != resultado(p) )
              ENTÃO
                     FF = FF – {p};
                     F = F + {p};

                     id_conj = NULO;
                     PARA (x começando em 2)
                     ENQUANTO ( x <= result-set-list.id_maior_conjunto
                                 E id_conj == NULO )
                     FAÇA
                            SE ( resultado(um nodo result-set-list[conjunto x])
                                == resultado(p) )
                            ENTÃO
                                   id_conj = x;
                            FIM_SE
                     FIM_PARA

                     SE ( id_conj != NULO )
                     ENTÃO
                            result-set-list[conjunto id_conj] =
                                   result-set-list[conjunto id_conj] + {p};
                     SENÃO
                            id_novo_conj = result-set-list.criar_novo_conjunto;
                            result-set-list[conjunto id_novo_conj] = {p};
                     FIM_SE
              FIM_SE
       ATÉ ( foi encontrado o estado de todos os nodos )

FIM_REPETIR
```

3.2.5 Diagnóstico Dinâmico e Procedimento de Recuperação

O algoritmo *Hi-Dif* possui a capacidade de realizar o diagnóstico de eventos dinâmicos, ou seja, eventos que ocorrem antes do diagnóstico de eventos anteriores por todos os nodos sem-falha do sistema. Em contraposição, o modelo de falhas estático adotado por alguns algoritmos hierárquicos de diagnóstico apresentados baseados no modelo PMC, considera que um evento não ocorre antes que todos os nodos sem-falha tenham diagnosticado o evento anterior. Entretanto, este modelo estático não é adequado para um ambiente real, onde falhas podem ocorrer mesmo antes de outras falhas ainda não terem sido diagnosticadas por todos os nodos sem-falha. Considerando-se um modelo de falhas dinâmico é necessário especificar um procedimento de recuperação para os nodos que são reparados durante a execução do algoritmo.

Quando um nodo *i* é reparado, este não tem informações de diagnóstico sobre o sistema. À medida que este nodo realiza seus testes e obtém informação de diagnóstico através de outros nodos do sistema, ele atualiza suas informações de diagnóstico. Neste momento é necessário distinguir quais informações de diagnóstico que este nodo *i* já possui estão atualizadas das não atualizadas, para impedir que informações incorretas sejam propagas pelo sistema.

Como cada nodo do sistema mantém informações (sobre o estado, conjunto na *result-set-list*, e *timestamp*) sobre todos os nodos do sistema, um campo chamado *u-bit* pode ser associado a estas informações. Este campo será utilizado para indicar se a informação de diagnóstico sobre um determinado nodo está atualizada ou não desde a

reparação do nodo testador. Este campo pode ser implementado como 1 bit, onde o valor 0 indica que a informação ainda não está atualizada e 1, no caso contrário. Um nodo que já completou o procedimento de recuperação, é um nodo que já possui informações atualizadas sobre todos os nodos do sistema, ou seja, um nodo que possui os *u-bits* de todos os nodos com o valor 1.

Ao iniciar o procedimento de recuperação, o nodo *i* atribui a todos os seus *u-bits* o valor 0. Ao encontrar um nodo sem-falha *j* que já tenha completado o processo de recuperação ele obtém informações – sobre o estado e sobre o valor do *timestamp* – de todos os nodos do sistema, inclusive sobre o seu próprio *timestamp* (o *timestamp* do nodo *i*). Em seguida este nodo *i* atribui a todos os *u-bits* de todos os nodos o valor 1. Este é o procedimento de recuperação que também já foi especificado no algoritmo *Hi-ADSD with Timestamps* em [53]. Como exemplo, a figura 3.7 mostra um sistema de 8 nodos no momento em que o nodo 0 se recupera de uma falha. Neste momento, todos os outros nodos do sistema já estavam sem-falha. Neste exemplo o nodo 0 inicia seus testes e como o nodo 1 está sem-falha, o nodo 0 obtém informação sobre todos os nodos do sistema a partir deste nodo 1.

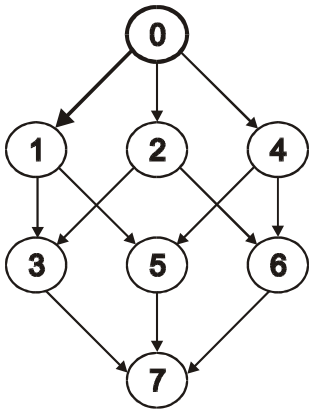

Figura 3.7: Sistema de 8 nodos onde os nodos de 1 a 7 já estavam sem-falha quando o nodo 0 torna-se sem-falha.

Uma simplificação que poderia ser considerada sobre o procedimento de recuperação especificado, é a não utilização de um *u-bit* que cada nodo testador mantém sobre todos os outros nodos, mas sim utilização de apenas um valor para a identificação de que o procedimento de recuperação já está completo neste nodo testador. Esta simplificação poderia ser feita, pois sempre que um nodo que está executando o procedimento encontra um nodo sem-falha, são obtidas informações sobre todos os nodos do sistema, atribuindo todos os *u-bits* com o valor 1. Caso não seja encontrado nenhum nodo sem-falha, ao final da rodada de testes, o nodo testador também atribui o valor 1 a todos os seus *u-bits*. Assim, sempre para um nodo testador, os valores dos *u-bits* que ele mantém sobre todos os nodos, estão todos em 1 ou 0. Apesar desta simplificação ser possível, no algoritmo *Hi-Dif* esta implementação não foi utilizada.

Neste trabalho foi detectada uma situação em que, sobre o procedimento de recuperação acima especificado, é necessário o tratamento de um caso excepcional para que o sistema não fique com informações inconsistentes. Este caso acontece quando o nodo *i* encontra um nodo sem-falha *j* que já tenha completado o processo de recuperação, mas este nodo *j*, ainda não detectou a recuperação do nodo *i*. Neste caso, se o nodo *i* atualizar o seu *timestamp* com o valor que o nodo *j* tem sobre o *timetamp* de *i*, o nodo *i* ficará com seu *timestamp* desatualizado. Isto acontece, pois no próximo teste que o nodo *j* realizar sobre o nodo *i*, o nodo *j* irá detectar a recuperação do nodo *i* e incrementar o *timestamp* que mantém sobre este nodo, ficando com uma unidade a mais em relação ao *timestamp* de *i* sobre si próprio. Para tratar este caso, a solução adotada foi, quando o nodo *i* recebe informações sobre todos os nodos do sistema do nodo *j*, o nodo *i* deve comparar seu estado com o estado que o nodo *j* tem sobre o nodo *i*. Caso o

estado seja o mesmo, ou seja, sem-falha, o nodo *i* apenas atualiza seu *timestamp*, caso contrário, o nodo *i* deve incrementar o *timestamp* recebido em uma unidade.

Como exemplo, a figura 3.8 ilustra um sistema de 8 nodos onde somente os nodos 0 e 7 estão sem-falha, quando o nodo 6 se recupera. Neste momento os nodos 0 e 7 ainda não possuem a informação da recuperação do nodo 6 e seus *timestamps* sobre o nodo 6 ainda possuem, por exemplo, o valor 3. Como o nodo 6 inicia seus testes, e seu único filho sem falha é o nodo 7, ele obtém informação sobre o estado e *timestamp* de todos os nodos do sistema. Neste momento, o nodo 6 identifica que o nodo 7 possui a informação de que o nodo 6 está falho, e então o nodo 6 incrementa o *timestamp* recebido sobre si, atualizando seu *timestamp* com o valor 4.

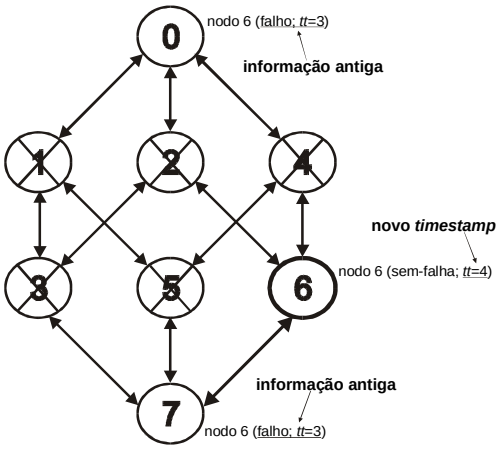

Figura 3.8: Sistema de 8 nodos onde o nodo 0 e 7 estão sem-falha quando ocorre a recuperação do nodo 6.

Assim como na recuperação de uma falha ou na inicialização de algoritmo, este procedimento de recuperação também é executado no algoritmo *Hi-Dif* quando um nodo detecta que seu próprio estado mudou, isto é, quando um nodo detecta uma mudança no seu próprio conteúdo. A execução do procedimento de recuperação neste

caso é necessária porque sempre que acontece uma modificação no conteúdo de um nodo, este nodo poderá começar a classificar como sem-falha, um novo conjunto de nodos. Desta forma, os valores dos *timestamps* que ele possui, podem ter sido obtidos com base em nodos que agora podem estar sendo classificados como nodos falhos. Portanto, o nodo que acabou de sofrer alteração de seu conteúdo, não pode mais considerar como correto os valores de seus *timestamps*, e deve obrigatoriamente executar o procedimento de recuperação.

3.3 Provas

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

A prova para a latência do algoritmo *Hi-Dif* é similar à prova para a latência do algoritmo *Hi-Comp* e mostrada através do *Teorema 1*.

Teorema 1: *Todos os nodos sem-falha executando o algoritmo* Hi-Dif *necessitam de, no máximo,* $\log_2 N$ *rodadas de testes para atingir o diagnóstico completo do sistema.*

Prova:

Considere um novo evento que ocorre no nodo a*. Pela definição de rodada de testes, todos os filhos do nodo* a*, realizam o diagnóstico desse evento na primeira rodada de testes posterior ao evento. Considerando o grafo* $T_a(S)$*, ilustrado na figura 3.9, na primeira rodada de testes posterior ao evento, os filhos do nodo* a *realizam o diagnóstico do evento que ocorre em* a.

Já na segunda rodada de testes, os nodos que possuem distância de diagnóstico igual a 2 em relação ao nodo a *realizam o diagnóstico, recebendo informações dos nodos com distância igual a 1, ou testando o nodo* a *diretamente, caso os nodos com distância 1 estejam falhos. Na* $T_a(S)$*, ilustrada na figura 3.9, os nodos que são filhos de* a *realizam o diagnóstico do evento em questão, recebendo informações dos filhos de* a *ou testando o próprio nodo* a.

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

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

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

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

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

A latência máxima do algoritmo ocorre entre os nodos com maior distância de diagnóstico do sistema. Pela definição de um hipercubo [36, 37] e de distância de diagnóstico já apresentada, a maior distância de diagnóstico entre dois nodos no sistema é de $\log_2 N$. *Portanto a latência máxima do algoritmo é de* $\log_2 N$ *rodadas de testes.* □

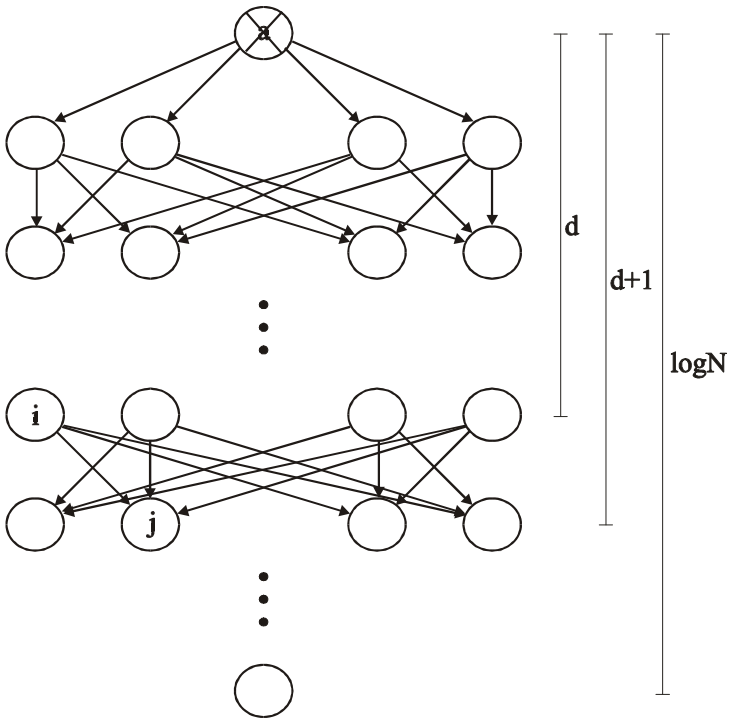

Figura 3.9: Ilustração de um grafo $T_a(S)$.

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

Prova:

Considerando um sistema com N *nodos, o pior caso para a quantidade de testes em uma rodada de testes é o pior caso de testes para cada um dos* N *nodos. No pior*

caso, a quantidade de testes necessárias para 1 *nodo ocorre quando este precisa testar todos os outros* N–1 *nodos do sistema; um exemplo ocorre na situação onde estes* N–1 *nodos estão falhos. Neste caso o nodo sem-falha executa* N–1 *testes.*

Considere este caso, em que há somente um nodo i *sem-falha no sistema e* N–1 *nodos falhos, mas não com falha do tipo crash. Também considere que cada um dos* N–1 *nodos está com o conteúdo replicado alterado de forma que todos os nodos estão com conteúdos diferentes entre si, ou seja, considerando os* N *nodos, tem-se* N *conteúdos diferentes e somente* 1 *correto.*

Para realizar o diagnóstico, o nodo sem-falha envia testes para si próprio e para cada um dos nodos falhos, então o número de testes realizados é N–1, *pois existem* N–1 *nodos falhos no sistema. O mesmo acontece com cada um dos outros* N–1 *nodos falhos. Como estão com falha que não é do tipo crash, cada nodo considera-se sem-falha e realiza também outros* N–1 *testes, pois para este nodo existem* N–1 *nodos falhos (nodos que estão com conteúdo diferente deste nodo testador). Ao considerar esta configuração do sistema, temos o pior caso do número de testes para cada nodo, ou seja, temos a seguinte quantidade de testes:* $N * (N–1) = N^2 – N$, *que é* $O(N^2)$. □

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

Prova:

Inicialmente considere um sistema com somente um nodo sem-falha e N–1 *nodos falhos. O nodo sem-falha vai testar todos os nodos do sistema, enviando testes para si próprio e para cada um dos nodos falhos, identificando o estado de todos os nodos.*

Agora considere um sistema com mais de um nodo sem-falha. Um desses nodos sem-falha realiza testes até encontrar outro nodo sem-falha. Quando o testador

encontra um nodo sem-falha, o testador obtém informações de diagnóstico do nodo testado. Juntando as informações recebidas do nodo sem-falha testado com as informações coletadas pelos seus próprios testes, o nodo sem-falha realiza um diagnóstico completo do sistema.

Entretanto, se ocorrer a situação ilustrada pela figura 3.10, na qual o nodo a *obtém informações de diagnóstico sobre o nodo* c *através do nodo* b *e o nodo* b *obtém informações de diagnóstico sobre o nodo* c *através do nodo* a*, os nodos* a *e* b *não completariam o diagnóstico do sistema.*

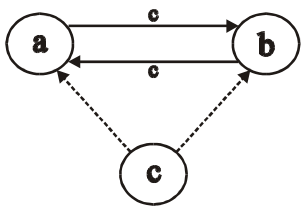

Figura 3.10: Caso impossível em um sistema executando o algoritmo *Hi-Dif*.

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

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

Capítulo 4

Resultados de Simulação

Neste capítulo são apresentados os resultados obtidos através da realização de simulação do algoritmo *Hi-Dif*. As simulações foram efetuadas utilizando-se a linguagem de simulação de eventos discretos SMPL (*SiMulation Programming Language*) [54]. Os nodos foram modelados como *facilities* do SMPL. Quatro tipos de eventos foram definidos: teste, falha, recuperação e alteração do conteúdo replicado.

Quatro tipos de experimentos foram simulados. As duas primeiras simulações mostram o funcionamento do algoritmo *Hi-Dif*. A primeira simulação mostra a realização do diagnóstico de um evento pelo algoritmo *Hi-Dif* em um sistema de 16 nodos na situação onde todos os nodos do sistema estão sem-falha quando ocorre um evento. A segunda simulação mostra a realização do diagnóstico quando temos todos os nodos falhos, exceto os nodos 0 e 15 que estão sem-falha, quando ocorre um evento no nodo 15.

Em seguida, a latência do algoritmo é apresentada com base também no primeiro experimento realizado. Na sequência, é avaliado o número máximo de testes necessários pelo algoritmo através de uma simulação em um sistema de 16 nodos onde 15 nodos

estão falhos. Também é feita a análise do número máximo de testes em um sistema de 16 nodos, mas considerando apenas falhas do tipo crash neste sistema. Para isso, são analisados os resultados de simulações para todas as combinações possíveis de falhas do tipo crash de nodos em um sistema de 16 nodos. Por fim, uma quarta simulação do algoritmo é realizada para, em conjunto com a segunda simulação, demonstrar a diagnosticabilidade (*diagnosability*) do sistema.

Por convenção, nas figuras de exemplos das simulações que serão descritas nas próximas seções, um nodo com falha do tipo crash será exemplificado como mostrado na figura 4.1 (a), e um nodo com falha ocasionada por uma alteração de seu conteúdo será exemplificado como mostrado na figura 4.1 (b).

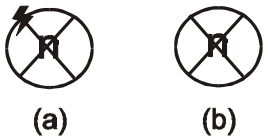

Figura 4.1: (a) Nodo com falha do tipo crash. (b) Nodo com falha devido à alteração de seu conteúdo.

4.1 Funcionamento do Algoritmo

O funcionamento do algoritmo é detalhado através da realização de duas simulações. Ambas as simulações foram realizadas em um sistema com 16 nodos. Na primeira simulação todos os nodos estão sem-falha quando ocorre a falha do nodo 15. Na segunda simulação, tem-se o caso onde o nodo 0 e o nodo 15 estão sem-falha e o restante dos nodos estão com falha do tipo crash, quando ocorre uma alteração de conteúdo que deixa o nodo 15 falho.

4.1.1 Simulação de um Sistema com *N*–1 Nodos Sem-Falha

Esta simulação foi realizada em um sistema com 16 nodos onde todos os nodos estão sem-falha e ocorre um evento no nodo 15, onde este nodo torna-se falho. Esta situação do sistema é apresentada através da figura 4.2.

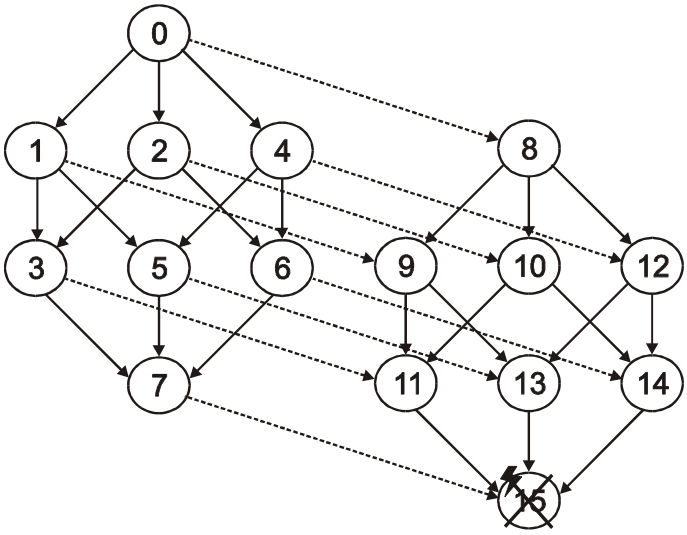

Figura 4.2: Sistema com 16 nodos onde o nodo 15 sofre falha do tipo crash.

O diagnóstico do sistema é analisado pela perspectiva do nodo 0, pois ele é o nodo mais distante do nodo 15 – nodo que sofre o evento – e será o último nodo a diagnosticar o evento. A seguir é mostrado um *logging* do resultado da simulação em fonte destacada. Neste *logging* são mostrados sempre os testes do nodo 0, e também os testes dos nodos que diagnosticam o novo evento ocorrido no nodo 15 a cada rodada de testes. Neste *logging* o nodo 15 fica falho no tempo 200, a próxima rodada de testes para todos os nodos ocorre no tempo 210 e cada rodada de testes ocorre sempre a cada

30 unidades de tempo, ou seja, se uma rodada de testes para um certo nodo ocorre no tempo 210, a próxima rodada de testes para este nodo ocorrerá no tempo 240.

Neste *logging*, sempre que se encontrar uma linha começando com '`FAULT:`' tem-se a indicação de que um nodo está ficando falho naquele momento. Sempre que se encontrar uma linha começando com '`Testing:`' tem-se a indicação de que os testes de um nodo estão começando. Sempre que existir uma linha começando com '`Test:`' tem-se a indicação de que um nodo está realizando um teste em um par de nodos, e nesta mesma linha estará a indicação do resultado da tarefa enviada aos dois nodos e também o resultado do diagnóstico do nodo testador sobre o nodo testado, ou seja, a classificação do nodo como falho (`faulty`) ou sem-falha (`ffree`), e também o número do conjunto na lista *result-set-list* em que este nodo será incluído (`conjunto x`). Sempre que se encontrar uma linha começando com '`End of Testing Round:`' tem-se a indicação de que os testes daquele nodo naquela rodada de testes acabaram e na sequência é mostrado o resultado do diagnóstico deste nodo testador sobre todos os nodos do sistema. O resultado do diagnóstico é mostrado com as seguintes informações para cada nodo, separadas por vírgulas: número do nodo, estado (falho ou sem-falha – `faulty` ou `ffree`), conjunto na lista result-set-list, o valor do conteúdo que este nodo possui, e valor do *timestamp* de cada nodo.

```
Simulação do Algoritmo HiDIF
Nodos = 16
Testing Interval =  30.0
Nodo a Falhar = 15
Fault Time = 200.0

FAULT: nodo 15 no tempo 200.0 torna-se falho (tipo crash).

Testing: nodo [0] no tempo 210.0 inicia testes.
  Test: nodo 0 tempo 210.0 testa nodos (0 e 1), result(100 e 100). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 2), result(100 e 100). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 4), result(100 e 100). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 8), result(100 e 100). DIAG: nodo 8 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,100,0  1,ffree,2,100,2  2,ffree,2,100,2  3,ffree,2,100,2
          4,ffree,2,100,2  5,ffree,2,100,2  6,ffree,2,100,2  7,ffree,2,100,2
          8,ffree,2,100,2  9,ffree,2,100,2  10,ffree,2,100,2  11,ffree,2,100,2
          12,ffree,2,100,2  13,ffree,2,100,2  14,ffree,2,100,2  15,ffree,2,100,2  >>
```

```
Testing: nodo [7] no tempo 210.0 inicia testes.
  Test: nodo 7 tempo 210.0 testa nodos (7 e 3), result(100 e 100). DIAG: nodo 3 é ffree (conjunto 1).
  Test: nodo 7 tempo 210.0 testa nodos (7 e 5), result(100 e 100). DIAG: nodo 5 é ffree (conjunto 1).
  Test: nodo 7 tempo 210.0 testa nodos (7 e 6), result(100 e 100). DIAG: nodo 6 é ffree (conjunto 1).
  Test: nodo 7 tempo 210.0 testa nodos (7 e 15), result(100 e 0). DIAG: nodo 15 é FAULTY (conjunto 0).
    End of Testing Round: diagnostico do nodo 7:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,0,0,3  >>
Testing: nodo [11] no tempo 210.0 inicia testes.
  Test: nodo 11 tempo 210.0 testa nodos (11 e 3), result(100 e 100). DIAG: nodo 3 é ffree (conjunto 1).
  Test: nodo 11 tempo 210.0 testa nodos (11 e 9), result(100 e 100). DIAG: nodo 9 é ffree (conjunto 1).
  Test: nodo 11 tempo 210.0 testa nodos (11 e 10), result(100 e 100). DIAG: nodo 10 é ffree (conjunto 1).
  Test: nodo 11 tempo 210.0 testa nodos (11 e 15), result(100 e 0). DIAG: nodo 15 é FAULTY (conjunto 0).
    End of Testing Round: diagnostico do nodo 11:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,0,0,3  >>
Testing: nodo [13] no tempo 210.0 inicia testes.
  Test: nodo 13 tempo 210.0 testa nodos (13 e 5), result(100 e 100). DIAG: nodo 5 é ffree (conjunto 1).
  Test: nodo 13 tempo 210.0 testa nodos (13 e 9), result(100 e 100). DIAG: nodo 9 é ffree (conjunto 1).
  Test: nodo 13 tempo 210.0 testa nodos (13 e 12), result(100 e 100). DIAG: nodo 12 é ffree (conjunto 1).
  Test: nodo 13 tempo 210.0 testa nodos (13 e 15), result(100 e 0). DIAG: nodo 15 é FAULTY (conjunto 0).
    End of Testing Round: diagnostico do nodo 13:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,0,0,3  >>
Testing: nodo [14] no tempo 210.0 inicia testes.
  Test: nodo 14 tempo 210.0 testa nodos (14 e 6), result(100 e 100). DIAG: nodo 6 é ffree (conjunto 1).
  Test: nodo 14 tempo 210.0 testa nodos (14 e 10), result(100 e 100). DIAG: nodo 10 é ffree (conjunto 1).
  Test: nodo 14 tempo 210.0 testa nodos (14 e 12), result(100 e 100). DIAG: nodo 12 é ffree (conjunto 1).
  Test: nodo 14 tempo 210.0 testa nodos (14 e 15), result(100 e 0). DIAG: nodo 15 é FAULTY (conjunto 0).
    End of Testing Round: diagnostico do nodo 14:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,0,0,3  >>

Testing: nodo [0] no tempo 240.0 inicia testes.
  Test: nodo 0 tempo 240.0 testa nodos (0 e 1), result(100 e 100). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 2), result(100 e 100). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 4), result(100 e 100). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 8), result(100 e 100). DIAG: nodo 8 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,100,0  1,ffree,2,100,2  2,ffree,2,100,2  3,ffree,2,100,2
          4,ffree,2,100,2  5,ffree,2,100,2  6,ffree,2,100,2  7,ffree,2,100,2
          8,ffree,2,100,2  9,ffree,2,100,2  10,ffree,2,100,2  11,ffree,2,100,2
          12,ffree,2,100,2  13,ffree,2,100,2  14,ffree,2,100,2  15,ffree,2,100,2  >>
Testing: nodo [3] no tempo 240.0 inicia testes.
  Test: nodo 3 tempo 240.0 testa nodos (3 e 1), result(100 e 100). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 2), result(100 e 100). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 7), result(100 e 100). DIAG: nodo 7 é ffree (conjunto 1).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 11), result(100 e 100). DIAG: nodo 11 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 1, conteúdo 0).
    End of Testing Round: diagnostico do nodo 3:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,1,0,3  >>
Testing: nodo [5] no tempo 240.0 inicia testes.
  Test: nodo 5 tempo 240.0 testa nodos (5 e 1), result(100 e 100). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 4), result(100 e 100). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 7), result(100 e 100). DIAG: nodo 7 é ffree (conjunto 1).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 13), result(100 e 100). DIAG: nodo 13 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 1, conteúdo 0).
    End of Testing Round: diagnostico do nodo 5:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,1,0,3  >>
Testing: nodo [6] no tempo 240.0 inicia testes.
  Test: nodo 6 tempo 240.0 testa nodos (6 e 2), result(100 e 100). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 4), result(100 e 100). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 7), result(100 e 100). DIAG: nodo 7 é ffree (conjunto 1).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 14), result(100 e 100). DIAG: nodo 14 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 1, conteúdo 0).
    End of Testing Round: diagnostico do nodo 6:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,1,0,3  >>
```

```
Testing: nodo [9] no tempo 240.0 inicia testes.
  Test: nodo 9 tempo 240.0 testa nodos (9 e 1), result(100 e 100). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 8), result(100 e 100). DIAG: nodo 8 é ffree (conjunto 1).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 11), result(100 e 100). DIAG: nodo 11 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 3, conteúdo 0).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 13), result(100 e 100). DIAG: nodo 13 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 9:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,3,0,3  >>
Testing: nodo [10] no tempo 240.0 inicia testes.
  Test: nodo 10 tempo 240.0 testa nodos (10 e 2), result(100 e 100). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 8), result(100 e 100). DIAG: nodo 8 é ffree (conjunto 1).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 11), result(100 e 100). DIAG: nodo 11 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 3, conteúdo 0).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 14), result(100 e 100). DIAG: nodo 14 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 10:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,0  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,3,0,3  >>
Testing: nodo [12] no tempo 240.0 inicia testes.
  Test: nodo 12 tempo 240.0 testa nodos (12 e 4), result(100 e 100). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 8), result(100 e 100). DIAG: nodo 8 é ffree (conjunto 1).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 13), result(100 e 100). DIAG: nodo 13 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 3, conteúdo 0).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 14), result(100 e 100). DIAG: nodo 14 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 12:
      <<  0,ffree,1,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,3,0,3  >>

Testing: nodo [0] no tempo 270.0 inicia testes.
  Test: nodo 0 tempo 270.0 testa nodos (0 e 1), result(100 e 100). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 0 tempo 270.0 testa nodos (0 e 2), result(100 e 100). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 0 tempo 270.0 testa nodos (0 e 4), result(100 e 100). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 0 tempo 270.0 testa nodos (0 e 8), result(100 e 100). DIAG: nodo 8 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,100,0  1,ffree,2,100,2  2,ffree,2,100,2  3,ffree,2,100,2
          4,ffree,2,100,2  5,ffree,2,100,2  6,ffree,2,100,2  7,ffree,2,100,2
          8,ffree,2,100,2  9,ffree,2,100,2  10,ffree,2,100,2  11,ffree,2,100,2
          12,ffree,2,100,2  13,ffree,2,100,2  14,ffree,2,100,2  15,ffree,2,100,2  >>
Testing: nodo [1] no tempo 270.0 inicia testes.
  Test: nodo 1 tempo 270.0 testa nodos (1 e 0), result(100 e 100). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 1 tempo 270.0 testa nodos (1 e 3), result(100 e 100). DIAG: nodo 3 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 3, conteúdo 0).
  Test: nodo 1 tempo 270.0 testa nodos (1 e 5), result(100 e 100). DIAG: nodo 5 é ffree (conjunto 1).
  Test: nodo 1 tempo 270.0 testa nodos (1 e 9), result(100 e 100). DIAG: nodo 9 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 1:
      <<  0,ffree,2,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,3,0,3  >>
Testing: nodo [2] no tempo 270.0 inicia testes.
  Test: nodo 2 tempo 270.0 testa nodos (2 e 0), result(100 e 100). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 2 tempo 270.0 testa nodos (2 e 3), result(100 e 100). DIAG: nodo 3 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 3, conteúdo 0).
  Test: nodo 2 tempo 270.0 testa nodos (2 e 6), result(100 e 100). DIAG: nodo 6 é ffree (conjunto 1).
  Test: nodo 2 tempo 270.0 testa nodos (2 e 10), result(100 e 100). DIAG: nodo 10 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 2:
      <<  0,ffree,2,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,3,0,3  >>
Testing: nodo [4] no tempo 270.0 inicia testes.
  Test: nodo 4 tempo 270.0 testa nodos (4 e 0), result(100 e 100). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 4 tempo 270.0 testa nodos (4 e 5), result(100 e 100). DIAG: nodo 5 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 3, conteúdo 0).
  Test: nodo 4 tempo 270.0 testa nodos (4 e 6), result(100 e 100). DIAG: nodo 6 é ffree (conjunto 1).
  Test: nodo 4 tempo 270.0 testa nodos (4 e 12), result(100 e 100). DIAG: nodo 12 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 4:
      <<  0,ffree,2,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,3,0,3  >>
Testing: nodo [8] no tempo 270.0 inicia testes.
  Test: nodo 8 tempo 270.0 testa nodos (8 e 0), result(100 e 100). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 8 tempo 270.0 testa nodos (8 e 9), result(100 e 100). DIAG: nodo 9 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 3, conteúdo 0).
  Test: nodo 8 tempo 270.0 testa nodos (8 e 10), result(100 e 100). DIAG: nodo 10 é ffree (conjunto 1).
  Test: nodo 8 tempo 270.0 testa nodos (8 e 12), result(100 e 100). DIAG: nodo 12 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 8:
      <<  0,ffree,2,100,0  1,ffree,1,100,2  2,ffree,1,100,2  3,ffree,1,100,2
          4,ffree,1,100,2  5,ffree,1,100,2  6,ffree,1,100,2  7,ffree,1,100,2
          8,ffree,1,100,2  9,ffree,1,100,2  10,ffree,1,100,2  11,ffree,1,100,2
          12,ffree,1,100,2  13,ffree,1,100,2  14,ffree,1,100,2  15, FAULTY,3,0,3  >>
```

```
Testing: nodo [0] no tempo 300.0 inicia testes.
  Test: nodo 0 tempo 300.0 testa nodos (0 e 1), result(100 e 100). DIAG: nodo 1 é ffree (conjunto 1).
    obtendo info: nodo 15 é FAULTY (conjunto 3, conteúdo 0).
  Test: nodo 0 tempo 300.0 testa nodos (0 e 2), result(100 e 100). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 0 tempo 300.0 testa nodos (0 e 4), result(100 e 100). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 0 tempo 300.0 testa nodos (0 e 8), result(100 e 100). DIAG: nodo 8 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,100,0  1,ffree,2,100,2  2,ffree,2,100,2  3,ffree,2,100,2
          4,ffree,2,100,2  5,ffree,2,100,2  6,ffree,2,100,2  7,ffree,2,100,2
          8,ffree,2,100,2  9,ffree,2,100,2  10,ffree,2,100,2  11,ffree,2,100,2
          12,ffree,2,100,2  13,ffree,2,100,2  14,ffree,2,100,2  15, FAULTY,3,0,3  >>
```

Na primeira rodada de testes posterior ao evento (no tempo 210), os nodos que conseguem diagnosticar a falha do nodo 15 são os nodos dos quais o nodo 15 é filho, ou seja, os nodos que possuem distância de diagnóstico igual a 1 para com o nodo 15, isto é, os nodos 7, 11, 13 e 14. Nesta rodada de testes o nodo 0 não realiza o diagnóstico sobre o evento ocorrido no nodo 15 pois o nodo 0 só obtém informação de diagnóstico dos nodos que são seus filhos, ou seja, dos nodos 1, 2, 4 e 8, e nenhum destes nodos possui informação sobre o evento ocorrido no nodo 15.

Na segunda rodada de testes após o evento (no tempo 240), os nodos que conseguem diagnosticar a falha do nodo 15, são os nodos 3, 5, 6, 9, 10 e 12. Estes nodos são os nodos que possuem distância de diagnóstico igual a 2 para com o nodo 15. Estes nodos conseguem diagnosticar o evento ocorrido ao testarem seus filhos, ou seja, a informação é repassada ao serem testados os nodos 7, 11, 13 e 14, que já possuem informação sobre o estado do nodo 15.

Na terceira rodada de testes (no tempo 270), os nodos 1, 2, 4 e 8 conseguem diagnosticar a falha do nodo 15. Estes nodos são os nodos que possuem distância de diagnóstico até o nodo 15 igual a 3. Estes conseguem diagnosticar o evento pois recebem informação de diagnóstico através de seus filhos que já diagnosticaram o evento na rodada de testes anterior.

Na quarta rodada (no tempo 300), os nodos com distância de diagnóstico até o nodo 15 igual a 4 realizam o diagnóstico do evento ocorrido no nodo 15, ou seja, o nodo 15 consegue diagnosticar o evento ocorrido.

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

4.1.2 Simulação de um Sistema com 1 Nodo Sem-Falha

Esta simulação foi realizada em um sistema com 16 nodos onde existem N–2 nodos com falha do tipo crash e acontece mais um evento de falha. Os nodos falhos são os nodos de identificadores de 1 a 14, e os nodos 0 e 15 estão sem-falha. Então ocorre um evento no nodo 15, onde acontece uma alteração no conteúdo deste nodo, ou seja, para o sistema este nodo torna-se falho, pois seu conteúdo foi alterado em relação ao considerado correto. Esta situação é apresentada através da figura 4.3.

O diagnóstico do sistema é analisado pela perspectiva do nodo 0, pois ele é o nodo mais distante do nodo 15 – nodo que sofre o evento – e será o último nodo a diagnosticar o evento. A seguir é mostrado um *logging* do resultado da simulação em fonte destacada. Neste *logging* são mostrados os testes necessários ao nodo 0 para completar o diagnóstico do sistema. Neste *logging* os nodos com identificadores de 1 a 14 estão falhos e o nodo 15 sofre alteração de conteúdo, ou seja, torna-se falho no tempo 200. A próxima rodada de testes para todos os nodos ocorre no tempo 210.

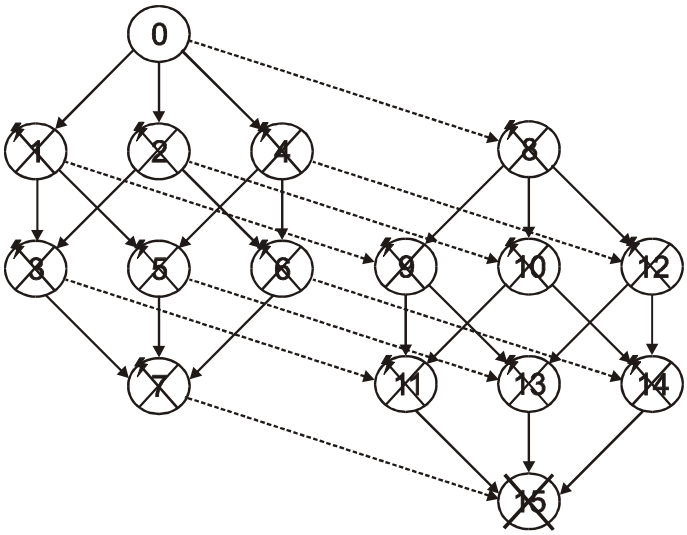

Figura 4.3: Sistema com 16 nodos onde todos os nodos de 1 a 14 estão falhos, e acontece uma alteração no conteúdo do nodo 15.

Neste *logging*, sempre que se encontrar uma linha começando com '`DATA MODIFICATION:`' tem-se a indicação de que um nodo está sendo alterado naquele momento. Sempre que se encontrar uma linha começando com '`Testing:`' tem-se a indicação de que os testes de um nodo estão começando. Sempre que existir uma linha começando com '`Test:`' tem-se a indicação de que um nodo está realizando um teste em um par de nodos, e nesta mesma linha estará a indicação do resultado da tarefa enviada aos dois nodos e também o resultado do diagnóstico do nodo testador sobre o nodo testado, ou seja, a classificação do nodo como falho (`faulty`) ou sem-falha (`ffree`), e também o número do conjunto na lista *result-set-list* em que este nodo será incluído (`conjunto x`). Sempre que se encontrar uma linha começando com '`End of Testing Round:`' tem-se a indicação de que os testes daquele nodo naquela rodada de testes acabaram e na sequência é mostrado o resultado do diagnóstico deste nodo testador sobre todos os nodos do sistema. O resultado do diagnóstico é mostrado com as seguintes informações para cada nodo, separadas por vírgulas: número do nodo, estado (falho ou

sem-falha – `faulty` ou `ffree`), conjunto na lista result-set-list, o valor do conteúdo que este nodo possui, e valor do *timestamp* de cada nodo.

```
Simulação do Algoritmo HiDIF
Nodos = 16
Testing Interval =  30.0
Nodo a Falhar = 15
Fault Time = 200.0

DATA MODIFICATION: nodo 15 no tempo 200.0 tem conteúdo alterado para 20.

Testing: nodo [0] no tempo 210.0 inicia testes.
  Test: nodo 0 tempo 210.0 testa nodos (0 e 1), result(100 e 0). DIAG: nodo 1 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 2), result(100 e 0). DIAG: nodo 2 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 4), result(100 e 0). DIAG: nodo 4 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 8), result(100 e 0). DIAG: nodo 8 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 3), result(100 e 0). DIAG: nodo 3 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 5), result(100 e 0). DIAG: nodo 5 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 6), result(100 e 0). DIAG: nodo 6 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 9), result(100 e 0). DIAG: nodo 9 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 10), result(100 e 0). DIAG: nodo 10 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 12), result(100 e 0). DIAG: nodo 12 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 7), result(100 e 0). DIAG: nodo 7 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 11), result(100 e 0). DIAG: nodo 11 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 13), result(100 e 0). DIAG: nodo 13 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 14), result(100 e 0). DIAG: nodo 14 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 15), result(100 e 20). DIAG: nodo 15 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,100,0  1,indef,0,0,2  2,indef,0,0,2  3,indef,0,0,2
          4,indef,0,0,2  5,indef,0,0,2  6,indef,0,0,2  7,indef,0,0,2
          8,indef,0,0,2  9,indef,0,0,2  10,indef,0,0,2  11,indef,0,0,2
          12,indef,0,0,2  13,indef,0,0,2  14,indef,0,0,2  15, FAULTY,2,20,3  >>
```

Então, na primeira rodada de testes posterior ao evento (no tempo 210), o nodo 0 inicia seus testes e testa cada um de seus filhos, ou seja, testa os nodos 1, 2, 4 e 8. Como todos estes nodos estão com falha do tipo crash, o nodo 0 não obtém nenhuma informação sobre nenhum outro nodo do sistema. Desta forma, o nodo 0 inicia os testes no restante dos nodos do sistema, e para cada nodo envia uma tarefa para este nodo e para si próprio, até testar o nodo 15. Quando o nodo 0 testa o nodo 15, descobre que seu conteúdo está alterado e classifica este nodo falho, colocando este nodo conteúdo no conjunto 2 da lista result-set-list, ou seja, coloca em um conjunto que não é o conjunto 1 dos nodos sem-falha, mas também não é o conjunto dos nodos com falha do tipo crash.

Desta forma pode-se notar que o nodo 0 diagnostica o estado de todos os nodos do sistema em apenas 1 rodada de testes, e são necessários N–1 testes nesta rodada de testes para se completar o diagnóstico do sistema.

4.2 Latência do Algoritmo

Para demonstrar a latência do algoritmo *Hi-Dif* no diagnóstico de um evento, foi utilizada uma simulação onde todos os nodos do sistema estão sem-falha quando ocorre a falha de um dos nodos. Para mostrar a latência do algoritmo, é utilizada a mesma simulação apresentada na seção 4.1.1 onde se tem um sistema de 16 nodos com todos os nodos sem-falha. Então o nodo 15 torna-se falho com falha do tipo crash. Esta simulação é representada pela figura 4.2.

Nesta simulação, ocorre a falha do nodo 15 e a informação sobre este evento deve ser propagada até atingir o nodo 0, que é o nodo mais distante do nodo onde o evento ocorre. Na primeira rodada de testes, os nodos dos quais o nodo 15 é filho realizam o diagnóstico, ou seja, os nodos 7, 11, 13 e 14. Na segunda rodada de testes posterior ao evento, os nodos dos quais os nodos 7, 11, 13 e 14 são filhos realizam o diagnóstico do evento, ou seja os nodos 3, 5, 6, 9, 10 e 12. Na terceira rodada de testes os pais destes nodos realizam o diagnóstico do sistema, ou seja, os nodos 1, 2, 4 e 8. Por fim, na quarta rodada de testes, o nodo 0 realiza o diagnóstico do evento ocorrido no nodo 15.

A tabela 4.1 abaixo mostra a quantidade de nodos que realizam o diagnóstico a cada rodada de testes. Na primeira rodada de testes posterior ao evento 4 nodos realizam o diagnóstico do evento. Na segunda rodada de testes 6 nodos realizam o diagnóstico. Na terceira rodada 4 nodos realizam o diagnóstico e na quarta rodada de testes somente um nodo realiza o diagnóstico.

Rodada de Testes	Nodos que Diagnosticam o Evento
1	4
2	6
3	4
4	1

Tabela 4.1: Quantidade de nodos em um sistema de 16 nodos que realizam o diagnóstico em cada rodada de testes.

Portanto, como mostrado no gráfico da figura 4.4, são necessárias 4 rodadas de testes para que todos os nodos sem-falha do sistema completem o diagnóstico do evento ocorrido no nodo 15. Este gráfico também mostra a quantidade de nodos que já diagnosticaram o novo evento a cada rodada de testes.

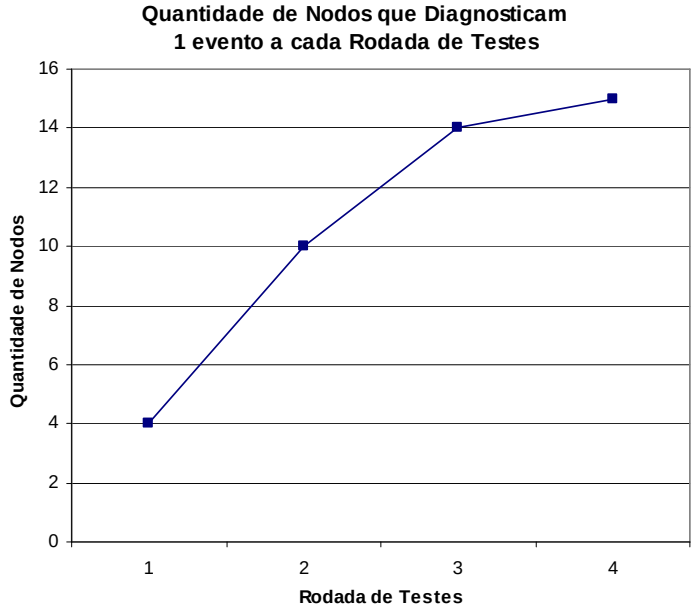

Figura 4.4: Quantidade de nodos que diagnosticam o evento em cada rodada de testes, em um sistema com N=16 nodos.

4.3 Quantidade Máxima de Testes Necessários

O propósito desta seção é mostrar a quantidade máxima de testes necessárias pelos nodos sem-falha em uma rodada de testes. Além disso, esta seção também mostra o pior caso para a quantidade de testes em um sistema de 16 nodos, quando consideradas somente falhas do tipo crash.

4.3.1 Pior Caso para o Número de Testes

Nesta seção é apresentada a simulação para uma sistema com 16 nodos onde temos somente um nodo sem-falha e os outros *N*–1 nodos estão falhos, mas com falha causada por alteração de conteúdo e, ainda, todos estes *N*–1 nodos estão com conteúdo diferentes entre si. Esta simulação exemplifica o pior caso de testes necessários, assim como no caso explicado através da prova do teorema 2. Esta situação do sistema é apresentada através da figura 4.5.

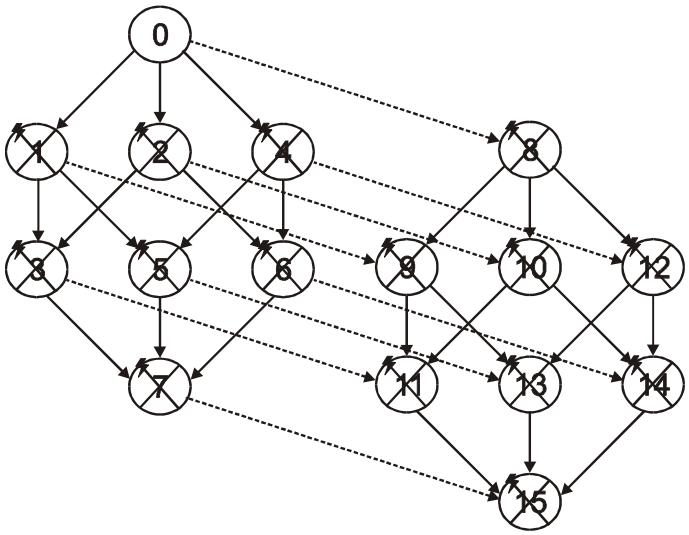

Figura 4.5: Sistema com 16 nodos somente o nodo 0 está sem falha e os outros N–1 nodos estão com falha causada por alteração de conteúdo.

A seguir é mostrado um *logging* do resultado da simulação em fonte destacada. Neste *logging* são mostrado todos os testes necessários para os 16 nodos do sistema na rodada de testes de número 240.

Neste *logging*, sempre que se encontrar uma linha começando com '`Testing:`' tem-se a indicação de que os testes de um nodo estão começando. Sempre que existir uma linha começando com '`Test:`' tem-se a indicação de que um nodo está realizando um teste em outros dois nodos, e nesta mesma linha estará a indicação do resultado da tarefa enviada aos dois nodos e também o resultado do diagnóstico do nodo testador sobre o nodo testado, ou seja, a classificação do nodo como falho (`faulty`) ou sem-falha (`ffree`), e também o número do conjunto na lista *result-set-list* em que este nodo será incluído (`conjunto x`).

```
Simulação do Algoritmo HiDIF
Nodos = 16
Testing Interval =  30.0

Testing: nodo [0] no tempo 240.0 inicia testes.
  Test: nodo 0 tempo 240.0 testa nodos (0 e 1), result(100 e 101). DIAG: nodo 1 é FAULTY (conjunto 2).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 2), result(100 e 102). DIAG: nodo 2 é FAULTY (conjunto 3).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 4), result(100 e 104). DIAG: nodo 4 é FAULTY (conjunto 4).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 8), result(100 e 108). DIAG: nodo 8 é FAULTY (conjunto 5).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 3), result(100 e 103). DIAG: nodo 3 é FAULTY (conjunto 6).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 5), result(100 e 105). DIAG: nodo 5 é FAULTY (conjunto 7).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 6), result(100 e 106). DIAG: nodo 6 é FAULTY (conjunto 8).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 9), result(100 e 109). DIAG: nodo 9 é FAULTY (conjunto 9).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 10), result(100 e 110). DIAG: nodo 10 é FAULTY (conjunto 10).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 12), result(100 e 112). DIAG: nodo 12 é FAULTY (conjunto 11).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 7), result(100 e 107). DIAG: nodo 7 é FAULTY (conjunto 12).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 11), result(100 e 111). DIAG: nodo 11 é FAULTY (conjunto 13).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 13), result(100 e 113). DIAG: nodo 13 é FAULTY (conjunto 14).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 14), result(100 e 114). DIAG: nodo 14 é FAULTY (conjunto 15).
  Test: nodo 0 tempo 240.0 testa nodos (0 e 15), result(100 e 115). DIAG: nodo 15 é FAULTY (conjunto 16).
Testing: nodo [1] no tempo 240.0 inicia testes.
  Test: nodo 1 tempo 240.0 testa nodos (1 e 0), result(101 e 100). DIAG: nodo 0 é FAULTY (conjunto 2).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 3), result(101 e 103). DIAG: nodo 3 é FAULTY (conjunto 3).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 5), result(101 e 105). DIAG: nodo 5 é FAULTY (conjunto 4).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 9), result(101 e 109). DIAG: nodo 9 é FAULTY (conjunto 5).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 2), result(101 e 102). DIAG: nodo 2 é FAULTY (conjunto 6).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 4), result(101 e 104). DIAG: nodo 4 é FAULTY (conjunto 7).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 7), result(101 e 107). DIAG: nodo 7 é FAULTY (conjunto 8).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 8), result(101 e 108). DIAG: nodo 8 é FAULTY (conjunto 9).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 11), result(101 e 111). DIAG: nodo 11 é FAULTY (conjunto 10).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 13), result(101 e 113). DIAG: nodo 13 é FAULTY (conjunto 11).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 6), result(101 e 106). DIAG: nodo 6 é FAULTY (conjunto 12).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 10), result(101 e 110). DIAG: nodo 10 é FAULTY (conjunto 13).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 12), result(101 e 112). DIAG: nodo 12 é FAULTY (conjunto 14).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 15), result(101 e 115). DIAG: nodo 15 é FAULTY (conjunto 15).
  Test: nodo 1 tempo 240.0 testa nodos (1 e 14), result(101 e 114). DIAG: nodo 14 é FAULTY (conjunto 16).
```

```
Testing: nodo [2] no tempo 240.0 inicia testes.
  Test: nodo 2 tempo 240.0 testa nodos (2 e 0), result(102 e 100). DIAG: nodo 0 é FAULTY (conjunto 2).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 3), result(102 e 103). DIAG: nodo 3 é FAULTY (conjunto 3).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 6), result(102 e 106). DIAG: nodo 6 é FAULTY (conjunto 4).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 10), result(102 e 110). DIAG: nodo 10 é FAULTY (conjunto 5).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 1), result(102 e 101). DIAG: nodo 1 é FAULTY (conjunto 6).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 4), result(102 e 104). DIAG: nodo 4 é FAULTY (conjunto 7).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 7), result(102 e 107). DIAG: nodo 7 é FAULTY (conjunto 8).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 8), result(102 e 108). DIAG: nodo 8 é FAULTY (conjunto 9).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 11), result(102 e 111). DIAG: nodo 11 é FAULTY (conjunto 10).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 14), result(102 e 114). DIAG: nodo 14 é FAULTY (conjunto 11).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 5), result(102 e 105). DIAG: nodo 5 é FAULTY (conjunto 12).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 9), result(102 e 109). DIAG: nodo 9 é FAULTY (conjunto 13).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 12), result(102 e 112). DIAG: nodo 12 é FAULTY (conjunto 14).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 15), result(102 e 115). DIAG: nodo 15 é FAULTY (conjunto 15).
  Test: nodo 2 tempo 240.0 testa nodos (2 e 13), result(102 e 113). DIAG: nodo 13 é FAULTY (conjunto 16).
Testing: nodo [3] no tempo 240.0 inicia testes.
  Test: nodo 3 tempo 240.0 testa nodos (3 e 1), result(103 e 101). DIAG: nodo 1 é FAULTY (conjunto 2).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 2), result(103 e 102). DIAG: nodo 2 é FAULTY (conjunto 3).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 7), result(103 e 107). DIAG: nodo 7 é FAULTY (conjunto 4).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 11), result(103 e 111). DIAG: nodo 11 é FAULTY (conjunto 5).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 0), result(103 e 100). DIAG: nodo 0 é FAULTY (conjunto 6).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 5), result(103 e 105). DIAG: nodo 5 é FAULTY (conjunto 7).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 6), result(103 e 106). DIAG: nodo 6 é FAULTY (conjunto 8).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 9), result(103 e 109). DIAG: nodo 9 é FAULTY (conjunto 9).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 10), result(103 e 110). DIAG: nodo 10 é FAULTY (conjunto 10).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 15), result(103 e 115). DIAG: nodo 15 é FAULTY (conjunto 11).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 4), result(103 e 104). DIAG: nodo 4 é FAULTY (conjunto 12).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 8), result(103 e 108). DIAG: nodo 8 é FAULTY (conjunto 13).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 13), result(103 e 113). DIAG: nodo 13 é FAULTY (conjunto 14).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 14), result(103 e 114). DIAG: nodo 14 é FAULTY (conjunto 15).
  Test: nodo 3 tempo 240.0 testa nodos (3 e 12), result(103 e 112). DIAG: nodo 12 é FAULTY (conjunto 16).
Testing: nodo [4] no tempo 240.0 inicia testes.
  Test: nodo 4 tempo 240.0 testa nodos (4 e 0), result(104 e 100). DIAG: nodo 0 é FAULTY (conjunto 2).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 5), result(104 e 105). DIAG: nodo 5 é FAULTY (conjunto 3).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 6), result(104 e 106). DIAG: nodo 6 é FAULTY (conjunto 4).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 12), result(104 e 112). DIAG: nodo 12 é FAULTY (conjunto 5).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 1), result(104 e 101). DIAG: nodo 1 é FAULTY (conjunto 6).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 2), result(104 e 102). DIAG: nodo 2 é FAULTY (conjunto 7).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 7), result(104 e 107). DIAG: nodo 7 é FAULTY (conjunto 8).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 8), result(104 e 108). DIAG: nodo 8 é FAULTY (conjunto 9).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 13), result(104 e 113). DIAG: nodo 13 é FAULTY (conjunto 10).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 14), result(104 e 114). DIAG: nodo 14 é FAULTY (conjunto 11).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 3), result(104 e 103). DIAG: nodo 3 é FAULTY (conjunto 12).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 9), result(104 e 109). DIAG: nodo 9 é FAULTY (conjunto 13).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 10), result(104 e 110). DIAG: nodo 10 é FAULTY (conjunto 14).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 15), result(104 e 115). DIAG: nodo 15 é FAULTY (conjunto 15).
  Test: nodo 4 tempo 240.0 testa nodos (4 e 11), result(104 e 111). DIAG: nodo 11 é FAULTY (conjunto 16).
Testing: nodo [5] no tempo 240.0 inicia testes.
  Test: nodo 5 tempo 240.0 testa nodos (5 e 1), result(105 e 101). DIAG: nodo 1 é FAULTY (conjunto 2).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 4), result(105 e 104). DIAG: nodo 4 é FAULTY (conjunto 3).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 7), result(105 e 107). DIAG: nodo 7 é FAULTY (conjunto 4).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 13), result(105 e 113). DIAG: nodo 13 é FAULTY (conjunto 5).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 0), result(105 e 100). DIAG: nodo 0 é FAULTY (conjunto 6).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 3), result(105 e 103). DIAG: nodo 3 é FAULTY (conjunto 7).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 6), result(105 e 106). DIAG: nodo 6 é FAULTY (conjunto 8).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 9), result(105 e 109). DIAG: nodo 9 é FAULTY (conjunto 9).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 12), result(105 e 112). DIAG: nodo 12 é FAULTY (conjunto 10).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 15), result(105 e 115). DIAG: nodo 15 é FAULTY (conjunto 11).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 2), result(105 e 102). DIAG: nodo 2 é FAULTY (conjunto 12).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 8), result(105 e 108). DIAG: nodo 8 é FAULTY (conjunto 13).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 11), result(105 e 111). DIAG: nodo 11 é FAULTY (conjunto 14).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 14), result(105 e 114). DIAG: nodo 14 é FAULTY (conjunto 15).
  Test: nodo 5 tempo 240.0 testa nodos (5 e 10), result(105 e 110). DIAG: nodo 10 é FAULTY (conjunto 16).
Testing: nodo [6] no tempo 240.0 inicia testes.
  Test: nodo 6 tempo 240.0 testa nodos (6 e 2), result(106 e 102). DIAG: nodo 2 é FAULTY (conjunto 2).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 4), result(106 e 104). DIAG: nodo 4 é FAULTY (conjunto 3).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 7), result(106 e 107). DIAG: nodo 7 é FAULTY (conjunto 4).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 14), result(106 e 114). DIAG: nodo 14 é FAULTY (conjunto 5).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 0), result(106 e 100). DIAG: nodo 0 é FAULTY (conjunto 6).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 3), result(106 e 103). DIAG: nodo 3 é FAULTY (conjunto 7).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 5), result(106 e 105). DIAG: nodo 5 é FAULTY (conjunto 8).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 10), result(106 e 110). DIAG: nodo 10 é FAULTY (conjunto 9).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 12), result(106 e 112). DIAG: nodo 12 é FAULTY (conjunto 10).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 15), result(106 e 115). DIAG: nodo 15 é FAULTY (conjunto 11).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 1), result(106 e 101). DIAG: nodo 1 é FAULTY (conjunto 12).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 8), result(106 e 108). DIAG: nodo 8 é FAULTY (conjunto 13).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 11), result(106 e 111). DIAG: nodo 11 é FAULTY (conjunto 14).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 13), result(106 e 113). DIAG: nodo 13 é FAULTY (conjunto 15).
  Test: nodo 6 tempo 240.0 testa nodos (6 e 9), result(106 e 109). DIAG: nodo 9 é FAULTY (conjunto 16).
Testing: nodo [7] no tempo 240.0 inicia testes.
  Test: nodo 7 tempo 240.0 testa nodos (7 e 3), result(107 e 103). DIAG: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 5), result(107 e 105). DIAG: nodo 5 é FAULTY (conjunto 3).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 6), result(107 e 106). DIAG: nodo 6 é FAULTY (conjunto 4).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 15), result(107 e 115). DIAG: nodo 15 é FAULTY (conjunto 5).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 1), result(107 e 101). DIAG: nodo 1 é FAULTY (conjunto 6).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 2), result(107 e 102). DIAG: nodo 2 é FAULTY (conjunto 7).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 4), result(107 e 104). DIAG: nodo 4 é FAULTY (conjunto 8).
```

```
  Test: nodo 7 tempo 240.0 testa nodos (7 e 11), result(107 e 111). DIAG: nodo 11 é FAULTY (conjunto 9).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 13), result(107 e 113). DIAG: nodo 13 é FAULTY (conjunto 10).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 14), result(107 e 114). DIAG: nodo 14 é FAULTY (conjunto 11).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 0), result(107 e 100). DIAG: nodo 0 é FAULTY (conjunto 12).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 9), result(107 e 109). DIAG: nodo 9 é FAULTY (conjunto 13).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 10), result(107 e 110). DIAG: nodo 10 é FAULTY (conjunto 14).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 12), result(107 e 112). DIAG: nodo 12 é FAULTY (conjunto 15).
  Test: nodo 7 tempo 240.0 testa nodos (7 e 8), result(107 e 108). DIAG: nodo 8 é FAULTY (conjunto 16).
Testing: nodo [8] no tempo 240.0 inicia testes.
  Test: nodo 8 tempo 240.0 testa nodos (8 e 0), result(108 e 100). DIAG: nodo 0 é FAULTY (conjunto 2).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 9), result(108 e 109). DIAG: nodo 9 é FAULTY (conjunto 3).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 10), result(108 e 110). DIAG: nodo 10 é FAULTY (conjunto 4).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 12), result(108 e 112). DIAG: nodo 12 é FAULTY (conjunto 5).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 1), result(108 e 101). DIAG: nodo 1 é FAULTY (conjunto 6).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 2), result(108 e 102). DIAG: nodo 2 é FAULTY (conjunto 7).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 4), result(108 e 104). DIAG: nodo 4 é FAULTY (conjunto 8).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 11), result(108 e 111). DIAG: nodo 11 é FAULTY (conjunto 9).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 13), result(108 e 113). DIAG: nodo 13 é FAULTY (conjunto 10).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 14), result(108 e 114). DIAG: nodo 14 é FAULTY (conjunto 11).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 3), result(108 e 103). DIAG: nodo 3 é FAULTY (conjunto 12).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 5), result(108 e 105). DIAG: nodo 5 é FAULTY (conjunto 13).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 6), result(108 e 106). DIAG: nodo 6 é FAULTY (conjunto 14).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 15), result(108 e 115). DIAG: nodo 15 é FAULTY (conjunto 15).
  Test: nodo 8 tempo 240.0 testa nodos (8 e 7), result(108 e 107). DIAG: nodo 7 é FAULTY (conjunto 16).
Testing: nodo [9] no tempo 240.0 inicia testes.
  Test: nodo 9 tempo 240.0 testa nodos (9 e 1), result(109 e 101). DIAG: nodo 1 é FAULTY (conjunto 2).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 8), result(109 e 108). DIAG: nodo 8 é FAULTY (conjunto 3).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 11), result(109 e 111). DIAG: nodo 11 é FAULTY (conjunto 4).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 13), result(109 e 113). DIAG: nodo 13 é FAULTY (conjunto 5).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 0), result(109 e 100). DIAG: nodo 0 é FAULTY (conjunto 6).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 3), result(109 e 103). DIAG: nodo 3 é FAULTY (conjunto 7).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 5), result(109 e 105). DIAG: nodo 5 é FAULTY (conjunto 8).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 10), result(109 e 110). DIAG: nodo 10 é FAULTY (conjunto 9).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 12), result(109 e 112). DIAG: nodo 12 é FAULTY (conjunto 10).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 15), result(109 e 115). DIAG: nodo 15 é FAULTY (conjunto 11).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 2), result(109 e 102). DIAG: nodo 2 é FAULTY (conjunto 12).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 4), result(109 e 104). DIAG: nodo 4 é FAULTY (conjunto 13).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 7), result(109 e 107). DIAG: nodo 7 é FAULTY (conjunto 14).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 14), result(109 e 114). DIAG: nodo 14 é FAULTY (conjunto 15).
  Test: nodo 9 tempo 240.0 testa nodos (9 e 6), result(109 e 106). DIAG: nodo 6 é FAULTY (conjunto 16).
Testing: nodo [10] no tempo 240.0 inicia testes.
  Test: nodo 10 tempo 240.0 testa nodos (10 e 2), result(110 e 102). DIAG: nodo 2 é FAULTY (conjunto 2).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 8), result(110 e 108). DIAG: nodo 8 é FAULTY (conjunto 3).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 11), result(110 e 111). DIAG: nodo 11 é FAULTY (conjunto 4).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 14), result(110 e 114). DIAG: nodo 14 é FAULTY (conjunto 5).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 0), result(110 e 100). DIAG: nodo 0 é FAULTY (conjunto 6).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 3), result(110 e 103). DIAG: nodo 3 é FAULTY (conjunto 7).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 6), result(110 e 106). DIAG: nodo 6 é FAULTY (conjunto 8).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 9), result(110 e 109). DIAG: nodo 9 é FAULTY (conjunto 9).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 12), result(110 e 112). DIAG: nodo 12 é FAULTY (conjunto 10).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 15), result(110 e 115). DIAG: nodo 15 é FAULTY (conjunto 11).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 1), result(110 e 101). DIAG: nodo 1 é FAULTY (conjunto 12).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 4), result(110 e 104). DIAG: nodo 4 é FAULTY (conjunto 13).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 7), result(110 e 107). DIAG: nodo 7 é FAULTY (conjunto 14).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 13), result(110 e 113). DIAG: nodo 13 é FAULTY (conjunto 15).
  Test: nodo 10 tempo 240.0 testa nodos (10 e 5), result(110 e 105). DIAG: nodo 5 é FAULTY (conjunto 16).
Testing: nodo [11] no tempo 240.0 inicia testes.
  Test: nodo 11 tempo 240.0 testa nodos (11 e 3), result(111 e 103). DIAG: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 9), result(111 e 109). DIAG: nodo 9 é FAULTY (conjunto 3).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 10), result(111 e 110). DIAG: nodo 10 é FAULTY (conjunto 4).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 15), result(111 e 115). DIAG: nodo 15 é FAULTY (conjunto 5).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 1), result(111 e 101). DIAG: nodo 1 é FAULTY (conjunto 6).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 2), result(111 e 102). DIAG: nodo 2 é FAULTY (conjunto 7).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 7), result(111 e 107). DIAG: nodo 7 é FAULTY (conjunto 8).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 8), result(111 e 108). DIAG: nodo 8 é FAULTY (conjunto 9).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 13), result(111 e 113). DIAG: nodo 13 é FAULTY (conjunto 10).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 14), result(111 e 114). DIAG: nodo 14 é FAULTY (conjunto 11).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 0), result(111 e 100). DIAG: nodo 0 é FAULTY (conjunto 12).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 5), result(111 e 105). DIAG: nodo 5 é FAULTY (conjunto 13).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 6), result(111 e 106). DIAG: nodo 6 é FAULTY (conjunto 14).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 12), result(111 e 112). DIAG: nodo 12 é FAULTY (conjunto 15).
  Test: nodo 11 tempo 240.0 testa nodos (11 e 4), result(111 e 104). DIAG: nodo 4 é FAULTY (conjunto 16).
Testing: nodo [12] no tempo 240.0 inicia testes.
  Test: nodo 12 tempo 240.0 testa nodos (12 e 4), result(112 e 104). DIAG: nodo 4 é FAULTY (conjunto 2).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 8), result(112 e 108). DIAG: nodo 8 é FAULTY (conjunto 3).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 13), result(112 e 113). DIAG: nodo 13 é FAULTY (conjunto 4).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 14), result(112 e 114). DIAG: nodo 14 é FAULTY (conjunto 5).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 0), result(112 e 100). DIAG: nodo 0 é FAULTY (conjunto 6).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 5), result(112 e 105). DIAG: nodo 5 é FAULTY (conjunto 7).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 6), result(112 e 106). DIAG: nodo 6 é FAULTY (conjunto 8).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 9), result(112 e 109). DIAG: nodo 9 é FAULTY (conjunto 9).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 10), result(112 e 110). DIAG: nodo 10 é FAULTY (conjunto 10).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 15), result(112 e 115). DIAG: nodo 15 é FAULTY (conjunto 11).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 1), result(112 e 101). DIAG: nodo 1 é FAULTY (conjunto 12).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 2), result(112 e 102). DIAG: nodo 2 é FAULTY (conjunto 13).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 7), result(112 e 107). DIAG: nodo 7 é FAULTY (conjunto 14).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 11), result(112 e 111). DIAG: nodo 11 é FAULTY (conjunto 15).
  Test: nodo 12 tempo 240.0 testa nodos (12 e 3), result(112 e 103). DIAG: nodo 3 é FAULTY (conjunto 16).
```

```
Testing: nodo [13] no tempo 240.0 inicia testes.
  Test: nodo 13 tempo 240.0 testa nodos (13 e 5), result(113 e 105). DIAG: nodo 5 é FAULTY (conjunto 2).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 9), result(113 e 109). DIAG: nodo 9 é FAULTY (conjunto 3).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 12), result(113 e 112). DIAG: nodo 12 é FAULTY (conjunto 4).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 15), result(113 e 115). DIAG: nodo 15 é FAULTY (conjunto 5).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 1), result(113 e 101). DIAG: nodo 1 é FAULTY (conjunto 6).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 4), result(113 e 104). DIAG: nodo 4 é FAULTY (conjunto 7).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 7), result(113 e 107). DIAG: nodo 7 é FAULTY (conjunto 8).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 8), result(113 e 108). DIAG: nodo 8 é FAULTY (conjunto 9).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 11), result(113 e 111). DIAG: nodo 11 é FAULTY (conjunto 10).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 14), result(113 e 114). DIAG: nodo 14 é FAULTY (conjunto 11).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 0), result(113 e 100). DIAG: nodo 0 é FAULTY (conjunto 12).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 3), result(113 e 103). DIAG: nodo 3 é FAULTY (conjunto 13).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 6), result(113 e 106). DIAG: nodo 6 é FAULTY (conjunto 14).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 10), result(113 e 110). DIAG: nodo 10 é FAULTY (conjunto 15).
  Test: nodo 13 tempo 240.0 testa nodos (13 e 2), result(113 e 102). DIAG: nodo 2 é FAULTY (conjunto 16).
Testing: nodo [14] no tempo 240.0 inicia testes.
  Test: nodo 14 tempo 240.0 testa nodos (14 e 6), result(114 e 106). DIAG: nodo 6 é FAULTY (conjunto 2).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 10), result(114 e 110). DIAG: nodo 10 é FAULTY (conjunto 3).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 12), result(114 e 112). DIAG: nodo 12 é FAULTY (conjunto 4).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 15), result(114 e 115). DIAG: nodo 15 é FAULTY (conjunto 5).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 2), result(114 e 102). DIAG: nodo 2 é FAULTY (conjunto 6).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 4), result(114 e 104). DIAG: nodo 4 é FAULTY (conjunto 7).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 7), result(114 e 107). DIAG: nodo 7 é FAULTY (conjunto 8).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 8), result(114 e 108). DIAG: nodo 8 é FAULTY (conjunto 9).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 11), result(114 e 111). DIAG: nodo 11 é FAULTY (conjunto 10).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 13), result(114 e 113). DIAG: nodo 13 é FAULTY (conjunto 11).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 0), result(114 e 100). DIAG: nodo 0 é FAULTY (conjunto 12).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 3), result(114 e 103). DIAG: nodo 3 é FAULTY (conjunto 13).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 5), result(114 e 105). DIAG: nodo 5 é FAULTY (conjunto 14).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 9), result(114 e 109). DIAG: nodo 9 é FAULTY (conjunto 15).
  Test: nodo 14 tempo 240.0 testa nodos (14 e 1), result(114 e 101). DIAG: nodo 1 é FAULTY (conjunto 16).
Testing: nodo [15] no tempo 240.0 inicia testes.
  Test: nodo 15 tempo 240.0 testa nodos (15 e 7), result(115 e 107). DIAG: nodo 7 é FAULTY (conjunto 2).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 11), result(115 e 111). DIAG: nodo 11 é FAULTY (conjunto 3).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 13), result(115 e 113). DIAG: nodo 13 é FAULTY (conjunto 4).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 14), result(115 e 114). DIAG: nodo 14 é FAULTY (conjunto 5).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 3), result(115 e 103). DIAG: nodo 3 é FAULTY (conjunto 6).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 5), result(115 e 105). DIAG: nodo 5 é FAULTY (conjunto 7).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 6), result(115 e 106). DIAG: nodo 6 é FAULTY (conjunto 8).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 9), result(115 e 109). DIAG: nodo 9 é FAULTY (conjunto 9).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 10), result(115 e 110). DIAG: nodo 10 é FAULTY (conjunto 10).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 12), result(115 e 112). DIAG: nodo 12 é FAULTY (conjunto 11).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 1), result(115 e 101). DIAG: nodo 1 é FAULTY (conjunto 12).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 2), result(115 e 102). DIAG: nodo 2 é FAULTY (conjunto 13).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 4), result(115 e 104). DIAG: nodo 4 é FAULTY (conjunto 14).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 8), result(115 e 108). DIAG: nodo 8 é FAULTY (conjunto 15).
  Test: nodo 15 tempo 240.0 testa nodos (15 e 0), result(115 e 100). DIAG: nodo 0 é FAULTY (conjunto 16).
```

Na primeira rodada de testes do *logging* (no tempo 240), o nodo 0 começa a testar seus filhos e, como todos estão com falha, ele testa todos os outros nodos que também estão com falha. Então nesta rodada de testes o nodo 0 realiza *N*–1 testes. Ao iniciar os testes do nodo 1, este nodo testa seus filhos. Como cada nodo testador sempre se considera sem-falha, apesar deste estar com conteúdo modificado do considerado correto, este nodo 1 diagnostica todos os seus filhos como nodos falhos, pois estes possuem conteúdo diferente do conteúdo do nodo 1. Então este nodo 1 testa todos os outros nodos do sistema onde também são diagnosticados como falhos. Então nesta rodada de testes o nodo 1 também realiza *N*–1 testes. O mesmo que aconteceu como o nodo 1, acontece com o nodo 2 e todos os outros nodos do sistema.

Portanto, nesta rodada de cada nodo realiza N–1 testes, ou seja, nesta rodada são executados N^2–N testes no sistema.

4.3.2 Pior Caso para o Número de Testes Considerando Falhas Tipo Crash

As simulações desta seção visam mostrar o pior caso para a quantidade de testes em um sistema de 16 nodos, quando consideradas somente falhas do tipo crash. Para a apresentação dos resultados foram realizadas simulações com todas as combinações de nodos possíveis para um sistema de 16 nodos, e escolhidas as que resultaram no maior número de testes necessários para todas as quantidades de nodos sem-falha. Para isso foram realizadas todas as simulações possíveis, com a quantidade de nodos sem-falha no sistema começando com 1, e incrementando este número em 1 até que os N nodos do sistema estejam sem-falha. Então foram selecionadas as situações onde os nodos sem-falha executam o maior número possível de testes.

Apesar de o valor apresentado para uma certa quantidade de nodos sem-falha ser o maior, podem existir outras combinações de nodos com esta mesma quantidade de nodos sem-falha que resulte na mesma quantidade de testes. Para a análise dos resultados, nesta seção não importa a combinação dos nodos, mas sim o número máximo de testes realizados pelos nodos.

A quantidade de testes necessários para cada quantidade de nodos sem-falha no sistema é mostrada através do gráfico da figura 4.6. Este gráfico mostra o número

máximo de testes necessários nas simulações do algoritmo considerando falhas do tipo crash para todas as quantidades de nodos falhos.

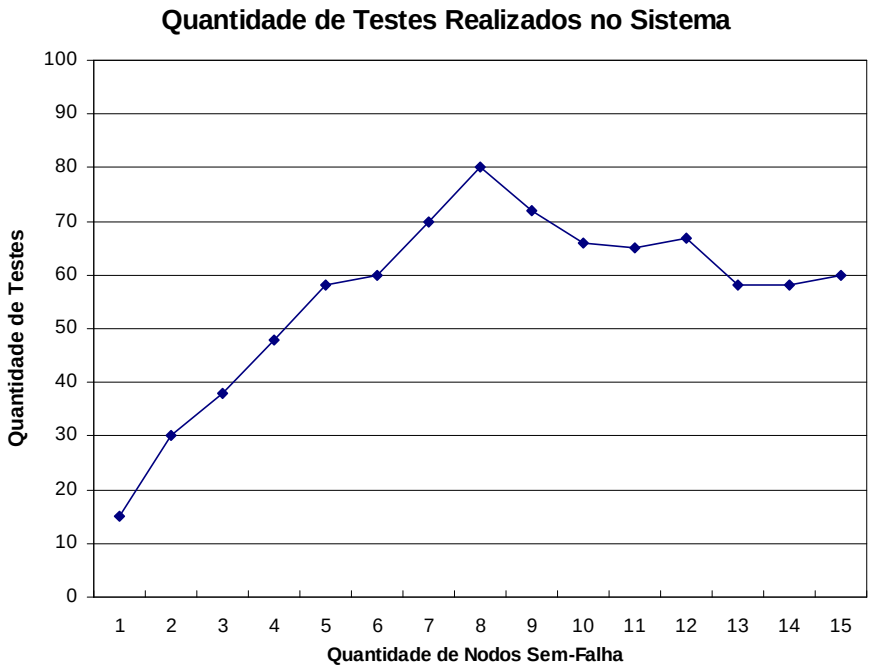

Figura 4.6: Número máximo de testes executados em um sistema de 16 nodos considerando somente falhas do tipo crash.

No gráfico da figura 4.6, nota-se que a maior quantidade de testes para os nodos sem-falha do sistema ocorre quando existem 8 nodos sem-falha, onde são necessários 80 testes no sistema para que todos os nodos completem o diagnóstico do sistema. Esse caso ocorre quando os nodos sem-falha estão dispostos como na figura 4.7.

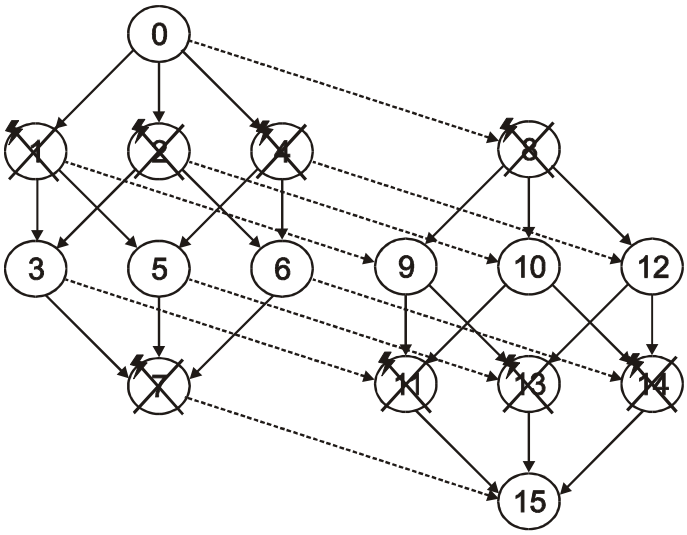

Figura 4.7: Pior caso do número de testes necessários em um sistema de 16 nodos.

No algoritmo *Hi-Dif*, um nodo testa seus filhos para obter informação de diagnóstico sobre os demais nodos do sistema, caso seus filhos estejam sem-falha. Quando seus filhos estão falhos, o nodo testador precisa testar os nodos filhos dos seus filhos e assim por diante. Como mostrado na figura 4.7, os estados dos nodos foram arranjados de forma que os filhos de cada nodo sem-falha estejam todos falhos. Assim são necessários, além dos 4 testes para testar seus filhos, mais 6 testes para testar os filhos dos seus filhos. Como temos 8 nodos sem-falha no sistema, são necessários 80 testes para que todos os nodos sem-falha completem o diagnóstico do sistema.

Essas simulações mostram que se temos muitos nodos com falha do tipo crash e o restante dos nodos sem-falha, o número de testes necessários em cada rodada de testes é bem abaixo do pior caso do algoritmo para o número máximo de testes que é $N^2 - N$, ou seja, no caso de um sistema com 16 nodos, 240 testes.

4.4 Diagnosticabilidade (*Diagnosability*)

A simulação “Simulação de um Sistema com *N*–1 Nodos Sem-Falha” apresentada na seção 4.1.1 comprova que um sistema executando o algoritmo *Hi-Dif* é (*N*–1)-diagnosticável. Nesta simulação temos os nodos 0 e 15 sem-falha e o restante dos nodos com falha do tipo crash quando ocorre a alteração do conteúdo do nodo 15, ou seja, quando este nodo fica falho. Nesta simulação, mesmo quando existe somente um nodo sem-falha no sistema, esse nodo consegue realizar o diagnostico de todos os *N*–1 nodos falhos.

Agora vamos considerar uma nova simulação em um sistema de 16 nodos onde todos os nodos estão sem-falha e, então, 15 nodos falham simulatenamente. A diferença desta simulação para a anterior é que na anterior existiam 14 nodos falhos e o nodo 15 fica falho em seguida resultando em 15 nodos falhos no sistema, já nesta nova simulação os 15 nodos ficam falhos no mesmo momento. Esta situação é apresentada através da figura 4.8.

O diagnóstico do sistema é analisado pela perspectiva do nodo 0, pois ele é único nodo sem-falha no sistema. A seguir é mostrado um *logging* do resultado da simulação em fonte destacada. Neste *logging* são mostrados os testes necessários ao nodo 0 para completar o diagnóstico do sistema.

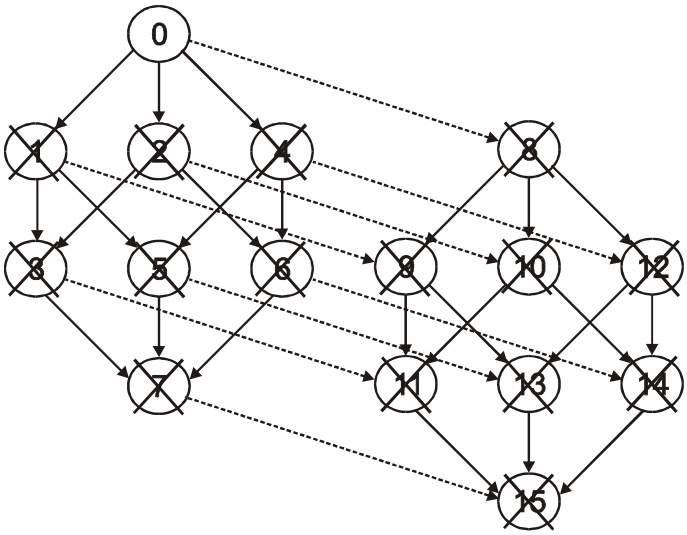

Figura 4.8: Sistema com 16 nodos onde todos os 1 a 15 estão falhos.

Neste *logging*, sempre que se encontrar uma linha começando com '`FAULT:`' tem-se a indicação de que um nodo fica falho naquele momento. Sempre que se encontrar uma linha começando com '`Testing:`' tem-se a indicação de que os testes de um nodo estão começando. Sempre que existir uma linha começando com '`Test:`' tem-se a indicação de que um nodo está realizando um teste em outros dois nodos, e nesta mesma linha estará a indicação do resultado da tarefa enviada aos dois nodos e, também, o resultado do diagnóstico do nodo testador sobre o nodo testado, ou seja, a classificação do nodo como falho (`faulty`) ou sem-falha (`ffree`), e também o número do conjunto na lista *result-set-list* em que este nodo será incluído (`conjunto x`). Sempre que se encontrar uma linha começando com '`End of Testing Round:`' tem-se a indicação de que os testes daquele nodo naquela rodada de testes acabaram e, na sequência, é mostrado o resultado do diagnóstico deste nodo testador sobre todos os nodos do sistema. O resultado do diagnóstico é mostrado com as seguintes informações para cada nodo, separadas por virgulas: número do nodo, estado (falho ou sem-falha – `faulty` ou `ffree`), conjunto na lista result-set-list, o valor do conteúdo que este nodo possui, e valor do *timestamp* de cada nodo.

```
Simulação do Algoritmo HiDIF
Nodos = 16
Testing Interval =  30.0
Nodo a Falhar = 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
Fault Time = 200.0

FAULT: nodo 1 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 2 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 3 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 4 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 5 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 6 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 7 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 8 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 9 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 10 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 11 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 12 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 13 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 14 no tempo 200.0 torna-se falho (tipo crash).
FAULT: nodo 15 no tempo 200.0 torna-se falho (tipo crash).

Testing: nodo [0] no tempo 210.0 inicia testes.
  Test: nodo 0 tempo 210.0 testa nodos (0 e 1), result(100 e 0). DIAG: nodo 1 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 2), result(100 e 0). DIAG: nodo 2 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 4), result(100 e 0). DIAG: nodo 4 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 8), result(100 e 0). DIAG: nodo 8 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 3), result(100 e 0). DIAG: nodo 3 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 5), result(100 e 0). DIAG: nodo 5 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 6), result(100 e 0). DIAG: nodo 6 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 9), result(100 e 0). DIAG: nodo 9 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 10), result(100 e 0). DIAG: nodo 10 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 12), result(100 e 0). DIAG: nodo 12 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 7), result(100 e 0). DIAG: nodo 7 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 11), result(100 e 0). DIAG: nodo 11 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 13), result(100 e 0). DIAG: nodo 13 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 14), result(100 e 0). DIAG: nodo 14 é FAULTY (conjunto 0).
  Test: nodo 0 tempo 210.0 testa nodos (0 e 15), result(100 e 0). DIAG: nodo 15 é FAULTY (conjunto 0).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,100,0  1, FAULTY,0,0,3  2, FAULTY,0,0,3  3, FAULTY,0,0,3
          4, FAULTY,0,0,3  5, FAULTY,0,0,3  6, FAULTY,0,0,3  7, FAULTY,0,0,3
          8, FAULTY,0,0,3  9, FAULTY,0,0,3  10, FAULTY,0,0,3  11, FAULTY,0,0,3
          12, FAULTY,0,0,3  13, FAULTY,0,0,3  14, FAULTY,0,0,3  15, FAULTY,0,0,3  >>
```

Na primeira rodada de testes posterior ao evento (no tempo 210), o nodo 0 inicia seus testes e testa cada um de seus filhos, ou seja, testa os nodos 1, 2, 4 e 8. Como todos estes nodos estão com falha do tipo crash, o nodo 0 não obtém nenhuma informação sobre nenhum outro nodo do sistema. Desta forma, o nodo 0 inicia os testes no restante dos nodos do sistema, e para cada nodo envia uma tarefa para este nodo e para si próprio, até testar o último nodo e diagnosticar a falha de todos estes nodos.

Portanto, pode-se constatar pelas duas simulações que, tanto na existência de um único evento ou com a existência de N–1 eventos simultâneos, o único nodo sem-falha no sistema consegue diagnosticar todos os eventos ocorridos no sistema. Portanto o algoritmo é N–1 diagnosticável.

Capítulo 5

Resultados Experimentais

Neste capítulo são apresentados os resultados obtidos através da realização de 4 experimentos práticos através da implementação real do algoritmo *Hi-Dif* para o diagnóstico de nodos com conteúdo replicado na Web. Todos os experimentos se passam em um sistema de 8 nodos. O primeiro experimento mostra a situação onde todos os nodos estão sem-falha quando acontece um evento no nodo 7 que sobre alteração de seu conteúdo. No segundo experimento, quando todos os nodos estão sem-falha, ocorre a falha do tipo crash de N–1 nodos. O terceiro experimento mostra a situação onde todos os nodos estão sem-falha quando acontece a alteração de conteúdo de $N/2$ nodos. No quarto experimento, os nodos 3 e 6 falham através da alteração do conteúdo replicado de forma diferente para cada um destes dois nodos.

Antes dos experimentos, que são apresentados a partir da seção 5.3, nas próximas duas seções são descritos detalhes sobre a implementação do algoritmo e uma descrição sobre o ambiente de realização dos experimentos.

5.1 Implementação do Algoritmo *Hi-Dif*

O algoritmo *Hi-Dif* foi implementado na linguagem de programação C [55], para ser utilizado em sistemas operacionais Linux [56]. A estratégia foi implementar o algoritmo através de dois componentes, ou dois executáveis: um servidor (*server-hidif*) e um cliente (*hidif*). O servidor é o componente responsável pela resposta aos testes ou informações solicitadas por outros nodos do sistema. O cliente é o componente no qual o algoritmo *Hi-Dif* está implementado. Este cliente é quem controla as rodadas de testes, realiza os testes, pede informações de diagnóstico para outros nodos do sistema e realiza o diagnóstico do sistema.

A comunicação entre os nodos do sistema como, por exemplo, os testes ou as solicitações de informações de diagnóstico sobre nodos, é realizada através de conexões TCP/IP (*Transmission Control Protocol/Internet Protocol*) [57] utilizando-se uma porta que pode ser configurada.

Tanto o servidor como o cliente utilizam um arquivo de configuração chamado ‘`hidif.ini`’. Neste arquivo de configuração estão indicados o número daquele nodo no sistema, o endereço IP (*Internet Protocol*) daquele nodo no sistema, a porta que ele utiliza para receber conexões TCP/IP, e o intervalo de testes entre cada rodada de testes indicado em segundos. Abaixo segue em fonte destacada, um exemplo de como são organizadas as informações neste arquivo de configuração para o nodo 0 de um sistema.

```
NODE 0
IP 192.168.254.1
PORT 7655
TEST_INTERVAL 10
```

Cada cliente do algoritmo *Hi-Dif* utiliza outro arquivo chamado ‘`hidif-nodes.ini`’ para guardar as configurações de cada nodo do sistema. Cada linha deste arquivo deve conter o identificador do nodo no sistema, o endereço IP do nodo e a porta que o algoritmo *Hi-Dif* implementa para receber conexões TCP/IP. Neste arquivo o primeiro nodo deve sempre começar com o identificador 0 e seus sucessores com uma unidade a mais em relação ao identificador da linha anterior. Este arquivo também deve conter as mesmas informações em todos os nodos do sistema. Abaixo segue em fonte destacada, um exemplo de como são organizadas as informações neste arquivo para um sistema com 8 nodos.

```
NODE_ID IP PORT
0 192.168.254.1 7655
1 192.168.254.2 7655
2 192.168.254.3 7655
3 192.168.254.4 7655
4 192.168.254.5 7655
5 192.168.254.6 7655
6 192.168.254.7 7655
7 192.168.254.8 7655
```

Quando um nodo *a* testa um par de nodos *b* e *c*, ou seja, os nodos *b* e *c* recebem uma tarefa para ser executada, cada um destes nodos executam um arquivo script de teste chamado ‘`script_test_hidif`’ que fica no diretório de instalação do algoritmo. Este script é livre e deve ser configurado por cada administrador do algoritmo especificamente para a sua aplicação. Este script deve obrigatoriamente gerar outro arquivo de resultado da tarefa chamado ‘`result_my_test.out`’ que deve conter o resultado da tarefa daquele nodo. Desta forma, quando um nodo recebe uma tarefa, o componente servidor executa o script de teste e responde para o nodo que solicitou a tarefa o conteúdo do arquivo de resultado da tarefa.

Cada nodo mantém um diretório chamado ‘data’. Neste diretório são mantidos arquivos que correspondem aos resultados das tarefas de cada nodo do sistema. Os arquivos deste diretório têm o nome de ‘`node_0.out`’ para o arquivo de resultado do nodo 0, ‘`node_1.out`’ para o arquivo de resultado do nodo 1, e assim por diante. Também neste diretório é mantido mais um arquivo chamado ‘`state_nodes.dat`’ que contém o resultado do diagnóstico daquele nodo sobre todos os nodos do sistema. Este arquivo é dinâmico, e a cada rodada de testes é atualizado com o resultado do diagnóstico do nodo testador.

Este arquivo que mantém o resultado do diagnóstico contém as seguintes informações: o identificador do nodo no sistema, o endereço IP do nodo, o estado do nodo (0 indica sem-falha e 1 indica que o nodo está falho), o valor do *timestamp* de cada nodo, e o conjunto na lista *result-set-list* em que cada nodo do sistema se encontra. Abaixo segue em fonte destacada, um exemplo de como são organizadas as informações neste arquivo para um sistema com 8 nodos. Neste exemplo nota-se que todos os nodos estão sem-falha exceto o nodo 7, e que este nodo 7 não está com falha do tipo crash, pois encontra-se no conjunto 2 da lista *result-set-list*, ou seja, está com o conteúdo modificado em relação ao considerado correto.

```
NODE_ID IP STATE TIMESTAMP RESULTSETLIST
0 192.168.254.1 0 0 1
1 192.168.254.1 0 2 1
2 192.168.254.1 0 2 1
3 192.168.254.1 0 2 1
4 192.168.254.1 0 2 1
5 192.168.254.1 0 2 1
6 192.168.254.1 0 2 1
7 192.168.254.1 1 3 2
```

Desta forma, esta implementação do algoritmo *Hi-Dif* possui algumas características: a execução de uma tarefa – recebida por uma conexão TCP/IP em porta configurável – foi implementada através da execução de 1 script pelo componente servidor e a saída desta é gerada em outro arquivo específico; e, como o componente cliente do algoritmo também grava o diagnóstico do sistema a cada rodada de testes em outro arquivo. Estas características permitem que o algoritmo (tanto o componente cliente quanto o servidor) possa até executar em outra máquina que não seja o servidor Web, caso não se queira incrementar a utilização de processamento dessas máquinas.

5.2 Ambiente de Realização dos Experimentos

Os experimentos foram realizados utilizando-se um computador com processador Intel Pentium IV [58] de 2400 MHz. O sistema operacional utilizado foi o Linux Kurumin [59] versão 2.20.

Todos os experimentos foram realizados em um sistema de 8 nodos e os experimentos foram todos realizados na mesma máquina. Os nodos dos experimentos foram configurados em diretórios diferentes nesta máquina, mas cada um utilizando uma porta diferente para as conexões TCP/IP. Desta forma, o arquivo com as configurações dos nodos '`hidif-nodes.ini`' que é o mesmo em todos os nodos, foi configurado como mostrada abaixo em fonte destacada.

```
NODE_ID IP PORT
0 192.168.254.1 7655
1 192.168.254.1 7656
2 192.168.254.1 7657
3 192.168.254.1 7658
4 192.168.254.1 7659
5 192.168.254.1 7660
6 192.168.254.1 7661
7 192.168.254.1 7662
```

Para ocasionar uma falha do tipo crash em um nodo, o processo servidor e o processo cliente do algoritmo *HiDif* foram terminados através do comando *kill* [56] do sistema operacional. Em todos os experimentos, os nodos foram configurados com um intervalo de 10 segundos de pausa entre cada rodada de testes, ou seja, se uma rodada de testes em um certo nodo terminou na data 23/07/2004 22:52:33, a próxima rodada de testes deve iniciar na data 23/07/2004 22:52:43.

Em todos os experimentos o conteúdo replicado monitorado foi um arquivo pdf de tamanho 3.591 KB. Para que este arquivo de pouco mais de 3,5 MB não fosse transferido pela rede a cada tarefa requisitada aos nodos e a utilização da rede não fosse prejudicada pela execução do algoritmo, o script de teste foi configurado para se executar o algoritmo de *hash* (resumo digital) MD5 [60] sobre o arquivo pdf, jogando apenas o *checksum* do MD5 no arquivo de resultado da tarefa. Desta forma o impacto no tráfego de rede ficou bastante reduzido.

Por convenção, nas figuras de exemplos dos experimentos que serão descritos neste capítulo, um nodo com falha do tipo crash será exemplificado como mostrado na figura 5.1 (a), e um nodo com falha ocasionada por uma alteração de seu conteúdo será exemplificado como mostrado na figura 5.1 (b).

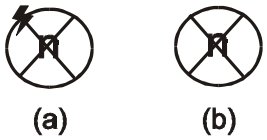

Figura 5.1: (a) Nodo com falha do tipo crash. (b) Nodo com falha devido à alteração de seu conteúdo.

Em cada um dos quatro experimentos descritos nas próximas seções, é apresentado um *logging*, onde neste *logging*, sempre que se encontrar uma linha começando com '`Testing:`' tem-se a indicação de que os testes de um nodo estão começando. Sempre que existir uma linha começando com '`Test:`' tem-se a indicação de que um nodo está realizando um teste em outros dois nodos, e nesta mesma linha estará a indicação do resultado da tarefa enviada aos dois nodos e também o resultado do diagnóstico do nodo testador sobre o nodo testado, ou seja, a classificado do nodo como falho (`faulty`) ou sem-falha (`ffree`), e também o número do conjunto na lista *result-set-list* em que este nodo será incluído (`conjunto x`). Sempre que se encontrar uma linha começando com '`End of Testing Round:`' tem-se a indicação de que os testes daquele nodo naquela rodada de testes acabaram e na sequência é mostrado o resultado do diagnóstico deste nodo testador sobre todos os nodos do sistema. O resultado do diagnóstico é mostrado com as seguintes informações para cada nodo, separadas por vírgulas: número do nodo, estado (falho ou sem-falha – `faulty` ou `ffree`), o conjunto na lista result-set-list, e o *timestamp* de cada nodo.

5.3 Primeiro Experimento – 1 Nodo Falha

O primeiro experimento mostra um sistema de 8 nodos onde todos os nodos estão sem-falha quando acontece um evento no nodo 7 que sofre alteração de seu conteúdo replicado. Esta situação do sistema é apresentada através da figura 5.2.

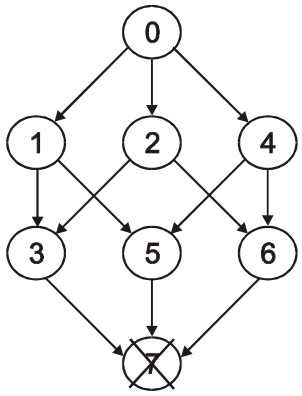

Figura 5.2: Um sistema de 8 nodos onde todos os nodos estão sem falha
e o nodo 7 sofre alteração de seu conteúdo.

A seguir é mostrado um *logging* do resultado do experimento em fonte destacada. Neste *logging* são mostrados todos os testes dos nodos sem-falha ocorridos após o evento, até que todos os nodos do sistema completem o diagnóstico deste evento. Neste *logging* o nodo 7 sofre alteração de conteúdo, ou seja, fica falho, no instante 27/07/2004 00:06:19.

Nodo 0

```
Testing: nodo [0] no tempo '27/07/2004 00:06:21' inicia testes.
  Test: nodo 0 tempo '27/07/2004 00:06:21' testa nodos (0 e 1). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 0 tempo '27/07/2004 00:06:21' testa nodos (0 e 2). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 0 tempo '27/07/2004 00:06:21' testa nodos (0 e 4). DIAG: nodo 4 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,ffree,1,2  >>
```

Nodo 1

```
Testing: nodo [1] no tempo '27/07/2004 00:06:21' inicia testes.
  Test: nodo 1 tempo '27/07/2004 00:06:22' testa nodos (1 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 1 tempo '27/07/2004 00:06:22' testa nodos (1 e 3). DIAG: nodo 3 é ffree (conjunto 1).
  Test: nodo 1 tempo '27/07/2004 00:06:22' testa nodos (1 e 5). DIAG: nodo 5 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 1:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,ffree,1,2  >>
```

Nodo 2

```
Testing: nodo [2] no tempo '27/07/2004 00:06:23' inicia testes.
  Test: nodo 2 tempo '27/07/2004 00:06:23' testa nodos (2 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 2 tempo '27/07/2004 00:06:23' testa nodos (2 e 3). DIAG: nodo 3 é ffree (conjunto 1).
  Test: nodo 2 tempo '27/07/2004 00:06:23' testa nodos (2 e 6). DIAG: nodo 6 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 2:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,ffree,1,2  >>
```

Nodo 3

```
Testing: nodo [3] no tempo '27/07/2004 00:06:23' inicia testes.
  Test: nodo 3 tempo '27/07/2004 00:06:23' testa nodos (3 e 1). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 3 tempo '27/07/2004 00:06:23' testa nodos (3 e 2). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 3 tempo '27/07/2004 00:06:23' testa nodos (3 e 7). DIAG: nodo 7 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 3:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,FAULTY,2,3  >>
```

Nodo 4

```
Testing: nodo [4] no tempo '27/07/2004 00:06:26' inicia testes.
  Test: nodo 4 tempo '27/07/2004 00:06:27' testa nodos (4 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 4 tempo '27/07/2004 00:06:27' testa nodos (4 e 5). DIAG: nodo 5 é ffree (conjunto 1).
  Test: nodo 4 tempo '27/07/2004 00:06:27' testa nodos (4 e 6). DIAG: nodo 6 é ffree (conjunto 1).
    obtendo info: nodo 7 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 4:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,FAULTY,2,3  >>
```

Nodo 5

```
Testing: nodo [5] no tempo '27/07/2004 00:06:26' inicia testes.
  Test: nodo 5 tempo '27/07/2004 00:06:27' testa nodos (5 e 1). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 5 tempo '27/07/2004 00:06:27' testa nodos (5 e 4). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 5 tempo '27/07/2004 00:06:27' testa nodos (5 e 7). DIAG: nodo 7 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 5:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,FAULTY,2,3  >>
```

Nodo 6

```
Testing: nodo [6] no tempo '27/07/2004 00:06:26' inicia testes.
  Test: nodo 6 tempo '27/07/2004 00:06:27' testa nodos (6 e 2). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 6 tempo '27/07/2004 00:06:27' testa nodos (6 e 4). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 6 tempo '27/07/2004 00:06:27' testa nodos (6 e 7). DIAG: nodo 7 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 6:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,FAULTY,2,3  >>
```

Nodo 0

```
Testing: nodo [0] no tempo '27/07/2004 00:06:31' inicia testes.
  Test: nodo 0 tempo '27/07/2004 00:06:32' testa nodos (0 e 1). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 0 tempo '27/07/2004 00:06:32' testa nodos (0 e 2). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 0 tempo '27/07/2004 00:06:32' testa nodos (0 e 4). DIAG: nodo 4 é ffree (conjunto 1).
    obtendo info: nodo 7 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,FAULTY,2,3  >>
```

Nodo 1

```
Testing: nodo [1] no tempo '27/07/2004 00:06:32' inicia testes.
  Test: nodo 1 tempo '27/07/2004 00:06:32' testa nodos (1 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 1 tempo '27/07/2004 00:06:32' testa nodos (1 e 3). DIAG: nodo 3 é ffree (conjunto 1).
  Test: nodo 1 tempo '27/07/2004 00:06:32' testa nodos (1 e 5). DIAG: nodo 5 é ffree (conjunto 1).
    obtendo info: nodo 7 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 1:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,FAULTY,2,3  >>
```

Nodo 2

```
Testing: nodo [2] no tempo '27/07/2004 00:06:33' inicia testes.
  Test: nodo 2 tempo '27/07/2004 00:06:33' testa nodos (2 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 2 tempo '27/07/2004 00:06:33' testa nodos (2 e 3). DIAG: nodo 3 é ffree (conjunto 1).
  Test: nodo 2 tempo '27/07/2004 00:06:33' testa nodos (2 e 6). DIAG: nodo 6 é ffree (conjunto 1).
    obtendo info: nodo 7 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 2:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,FAULTY,2,3  >>
```

Neste experimento, o primeiro nodo a realizar testes após o evento é o nodo 0 que inicia sua rodada de testes no instante 27/07/2004 00:06:21. Em seguida os nodos 1 e 2 realizam a próxima rodada de testes e também não detectam o evento. Até este momento, nenhum nodo detectou o evento ocorrido até que os nodos dos quais o nodo 7 é filho começam a executar seus testes. Nos instantes 27/07/2004 00:06:23, 27/07/2004 00:06:26 e 27/07/2004 00:06:26, os nos 3, 5 e 6 iniciam suas respectivas rodadas de testes. Como estes são os nodos dos quais o nodo 7 é filho (nodo onde o evento ocorreu), estes nodos detectam a falha do nodo 7 através de um teste direto a este nodo. Como o nodo 7 está falho, mas não com falha do tipo crash, este nodo é colocado no conjunto 2 da lista *result-set-list*.

Nos instantes 27/07/2004 00:06:27, 27/07/2004 00:06:32 e 27/07/2004 00:06:33, respectivamente os nodos 4, 1 e 2 diagnosticam o evento através de testes realizados sobre os nodos 5 e 6 que haviam a pouco tempo detectado o evento. A próxima rodada de testes do nodo 0 inicia-se no instante 27/07/2004 00:06:31. Assim, logo que o nodo 0 testa o nodo 4 – que é seu único filho que já detectou o evento neste instante – o nodo 0 obtém informação de diagnóstico do nodo 7 identificando a falha deste nodo.

5.4 Segundo Experimento – *N*–1 Nodos Falham

O segundo experimento mostra um sistema de 8 nodos onde todos os nodos estão sem-falha quando acontece a falha do tipo crash dos nodos com identificadores de 1 a 7, ou seja, acontecem *N*–1 falhas do tipo crash. Esta situação do sistema é apresentada através da figura 5.3.

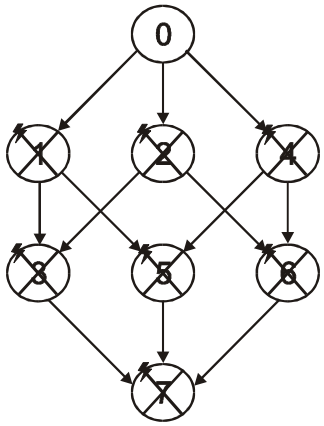

Figura 5.3: Um sistema de 8 nodos onde todos os nodos estão sem falha quando ocorre a falha do tipo crash dos nodos identificados de 1 a 7.

A seguir é mostrado um *logging* do resultado do experimento em fonte destacada. Neste *logging* são mostrados todos os testes do nodo 0 que é o único nodo sem-falha no sistema após a ocorrência dos eventos. Neste *logging* os nodos identificados de 1 a 7 ficam falhos no instante 27/07/2004 00:14:02.

Nodo 0

```
Testing: nodo [0] no tempo '27/07/2004 00:14:09' inicia testes.
  Test: nodo 0 tempo '27/07/2004 00:14:09' testa nodos (0 e 1). DIAG: nodo 1 é FAULTY (conjunto 0).
  Test: nodo 0 tempo '27/07/2004 00:14:09' testa nodos (0 e 2). DIAG: nodo 2 é FAULTY (conjunto 0).
  Test: nodo 0 tempo '27/07/2004 00:14:09' testa nodos (0 e 4). DIAG: nodo 4 é FAULTY (conjunto 0).
  Test: nodo 0 tempo '27/07/2004 00:14:09' testa nodos (0 e 3). DIAG: nodo 3 é FAULTY (conjunto 0).
  Test: nodo 0 tempo '27/07/2004 00:14:09' testa nodos (0 e 5). DIAG: nodo 5 é FAULTY (conjunto 0).
  Test: nodo 0 tempo '27/07/2004 00:14:09' testa nodos (0 e 6). DIAG: nodo 6 é FAULTY (conjunto 0).
  Test: nodo 0 tempo '27/07/2004 00:14:09' testa nodos (0 e 7). DIAG: nodo 7 é FAULTY (conjunto 0).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,0  1,FAULTY,0,5  2,FAULTY,0,5  3,FAULTY,0,7
          4,FAULTY,0,5  5,FAULTY,0,3  6,FAULTY,0,5  7,FAULTY,0,5  >>
```

Neste experimento, o nodo 0 é o único nodo sem-falha no sistema após as falhas do restante dos nodos. Desta forma, o nodo 0 precisa testar todos os nodos para se obter o diagnóstico do sistema. Assim, o nodo 0 realiza os N–1 = 7 testes e diagnostica todos os nodos outros como falhos colocando-os no conjunto 0 da lista *result-set-list*.

5.5 Terceiro Experimento – *N*/2 Nodos Sofrem Alteração de Conteúdo

O terceiro experimento mostra um sistema de 8 nodos onde todos os nodos estão sem-falha quando acontece um evento ao mesmo tempo nos nodos 1, 2, 3 e 4 que sofrem a mesma alterações no conteúdo replicado. Esta situação do sistema é apresentada através da figura 5.4.

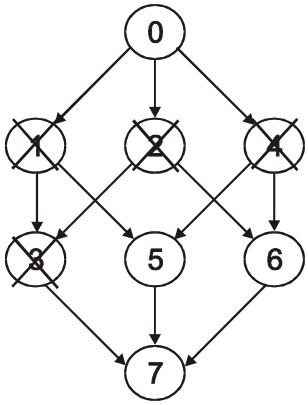

Figura 5.4: Um sistema de 8 nodos onde todos os nodos estão sem falha quando os nodos 1, 2, 3 e 4 sofrem a mesma alterações no conteúdo replicado.

A seguir é mostrado um *logging* do resultado do experimento em fonte destacada. Neste *logging* são mostrados todos os testes dos nodos sem-falha ocorridos após os quatro eventos, até que todos os nodos do sistema completem o diagnóstico deste dois eventos. Neste *logging* os nodos 1, 2, 3 e 4 sofrem alteração de conteúdo, ou seja, ficam falhos, no instante 27/07/2004 00:10:58.

Nodo 5

```
Testing: nodo [5] no tempo '27/07/2004 00:11:00' inicia testes (Rodada Teste: 32).
  Test: nodo 5 tempo '27/07/2004 00:11:00' testa nodos (5 e 1). DIAG: nodo 1 é FAULTY (conjunto 2).
  Test: nodo 5 tempo '27/07/2004 00:11:00' testa nodos (5 e 4). DIAG: nodo 4 é FAULTY (conjunto 2).
  Test: nodo 5 tempo '27/07/2004 00:11:01' testa nodos (5 e 7). DIAG: nodo 7 é ffree (conjunto 1).
  Test: nodo 5 tempo '27/07/2004 00:11:01' testa nodos (5 e 0). DIAG: nodo 0 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 5:
      <<  0,ffree,1,0  1,FAULTY,2,3  2,ffree,1,2  3,ffree,1,4
          4,FAULTY,2,3  5,ffree,1,2  6,ffree,1,4  7,ffree,1,4  >>
```

Nodo 6

```
Testing: nodo [6] no tempo '27/07/2004 00:11:02' inicia testes (Rodada Teste: 32).
  Test: nodo 6 tempo '27/07/2004 00:11:03' testa nodos (6 e 2). DIAG: nodo 2 é FAULTY (conjunto 2).
  Test: nodo 6 tempo '27/07/2004 00:11:03' testa nodos (6 e 4). DIAG: nodo 4 é FAULTY (conjunto 2).
  Test: nodo 6 tempo '27/07/2004 00:11:03' testa nodos (6 e 7). DIAG: nodo 7 é ffree (conjunto 1).
  Test: nodo 6 tempo '27/07/2004 00:11:03' testa nodos (6 e 0). DIAG: nodo 0 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 6:
      <<  0,ffree,1,0  1,ffree,1,2  2,FAULTY,2,3  3,ffree,1,4
          4,FAULTY,2,3  5,ffree,1,2  6,ffree,1,4  7,ffree,1,4  >>
```

Nodo 0

```
Testing: nodo [0] no tempo '27/07/2004 00:11:03' inicia testes (Rodada Teste: 33).
  Test: nodo 0 tempo '27/07/2004 00:11:03' testa nodos (0 e 1). DIAG: nodo 1 é FAULTY (conjunto 2).
  Test: nodo 0 tempo '27/07/2004 00:11:03' testa nodos (0 e 2). DIAG: nodo 2 é FAULTY (conjunto 2).
  Test: nodo 0 tempo '27/07/2004 00:11:03' testa nodos (0 e 4). DIAG: nodo 4 é FAULTY (conjunto 2).
  Test: nodo 0 tempo '27/07/2004 00:11:03' testa nodos (0 e 3). DIAG: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 0 tempo '27/07/2004 00:11:04' testa nodos (0 e 5). DIAG: nodo 5 é ffree (conjunto 1).
  Test: nodo 0 tempo '27/07/2004 00:11:04' testa nodos (0 e 6). DIAG: nodo 6 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,0  1,FAULTY,2,3  2,FAULTY,2,3  3,FAULTY,2,5
          4,FAULTY,2,3  5,ffree,1,2  6,ffree,1,4  7,ffree,1,4  >>
```

Nodo 7

```
Testing: nodo [7] no tempo '27/07/2004 00:11:06' inicia testes (Rodada Teste: 32).
  Test: nodo 7 tempo '27/07/2004 00:11:07' testa nodos (7 e 3). DIAG: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 7 tempo '27/07/2004 00:11:07' testa nodos (7 e 5). DIAG: nodo 5 é ffree (conjunto 1).
    obtendo info: nodo 1 é FAULTY (conjunto 2).
    obtendo info: nodo 4 é FAULTY (conjunto 2).
  Test: nodo 7 tempo '27/07/2004 00:11:07' testa nodos (7 e 6). DIAG: nodo 6 é ffree (conjunto 1).
    obtendo info: nodo 2 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 7:
      <<  0,ffree,1,0  1,FAULTY,2,3  2,FAULTY,2,3  3,FAULTY,2,5
          4,FAULTY,2,3  5,ffree,1,2  6,ffree,1,4  7,ffree,1,4  >>
```

Nodo 5

```
Testing: nodo [5] no tempo '27/07/2004 00:11:11' inicia testes (Rodada Teste: 33).
  Test: nodo 5 tempo '27/07/2004 00:11:11' testa nodos (5 e 1). DIAG: nodo 1 é FAULTY (conjunto 2).
  Test: nodo 5 tempo '27/07/2004 00:11:11' testa nodos (5 e 4). DIAG: nodo 4 é FAULTY (conjunto 2).
  Test: nodo 5 tempo '27/07/2004 00:11:11' testa nodos (5 e 7). DIAG: nodo 7 é ffree (conjunto 1).
    obtendo info: nodo 2 é FAULTY (conjunto 2).
    obtendo info: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 5 tempo '27/07/2004 00:11:11' testa nodos (5 e 0). DIAG: nodo 0 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 5:
      <<  0,ffree,1,0  1,FAULTY,2,3  2,FAULTY,2,3  3,FAULTY,2,5
          4,FAULTY,2,3  5,ffree,1,2  6,ffree,1,4  7,ffree,1,4  >>
```

Nodo 6

```
Testing: nodo [6] no tempo '27/07/2004 00:11:13' inicia testes (Rodada Teste: 33).
  Test: nodo 6 tempo '27/07/2004 00:11:14' testa nodos (6 e 2). DIAG: nodo 2 é FAULTY (conjunto 2).
  Test: nodo 6 tempo '27/07/2004 00:11:14' testa nodos (6 e 4). DIAG: nodo 4 é FAULTY (conjunto 2).
  Test: nodo 6 tempo '27/07/2004 00:11:14' testa nodos (6 e 7). DIAG: nodo 7 é ffree (conjunto 1).
    obtendo info: nodo 1 é FAULTY (conjunto 2).
    obtendo info: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 6 tempo '27/07/2004 00:11:15' testa nodos (6 e 0). DIAG: nodo 0 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 6:
      <<  0,ffree,1,0  1,FAULTY,2,3  2,FAULTY,2,3  3,FAULTY,2,5
          4,FAULTY,2,3  5,ffree,1,2  6,ffree,1,4  7,ffree,1,4  >>
```

Neste experimento, o primeiro nodo a realizar testes após os eventos é o nodo 5 que inicia sua rodada de testes no instante 27/07/2004 00:11:00. Então este nodo detecta as falhas dos nodos 1 e 4 pois testa diretamente estes dois nodos. Em seguida, o nodo 6 inicia sua rodada de testes e acontece uma situação parecida à ocorrida no nodo 5. O nodo 6 detecta a falha dos nodos 2 e 4 pois também testa diretamente estes dois nodos. O nodo 0 que inicia sua próxima rodada de testes no tempo 27/07/2004 00:11:03 detecta a falha dos quatro nodos, pois os nodos 1, 2 e 4 são seus e são testados diretamente pelo nodo 0, e o nodo 3 é testado na sequência após diagnosticar todos os seus filhos como falhos.

A seguir o nodo 7 inicia seus testes no tempo 27/07/2004 00:11:06 e consegue diagnosticar a falha de todos os nodos da seguinte maneira: a falha do nodo 3 é diagnosticada através de um teste, pois o nodo 3 é seu filho; as falhas dos nodos 1 e 4 são detectadas através de informações de diagnóstico recebidas através do nodo 5; e, a falha do nodo 2 é detectada através das informações de diagnóstico recebidas através do nodo 6. Na sequência, o nodo 5 diagnostica as falhas do nodo 2 e 3 através de informações recebidas do nodo 7, e o nodo 6 diagnostica as falhas do nodo 1 e 3 através de informações recebidas do nodo 7.

5.6 Quarto Experimento – 2 Nodos Sofrem Alterações Diferentes

O quarto experimento mostra um sistema de 8 nodos onde todos os nodos estão sem-falha quando acontece um evento ao mesmo tempo nos nodos 3 e 6 que sofrem alterações diferentes no conteúdo replicado. Esta situação do sistema é apresentada através da figura 5.5.

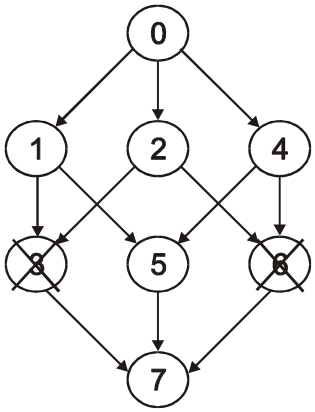

Figura 5.5: Um sistema de 8 nodos onde todos os nodos estão sem falha quando os nodos 3 e 6 sofrem alterações diferentes no conteúdo replicado.

A seguir é mostrado um *logging* do resultado do experimento em fonte destacada. Neste *logging* são mostrados todos os testes dos nodos sem-falha ocorridos após os dois eventos, até que todos os nodos do sistema completem o diagnóstico deste dois eventos. Neste *logging* os nodos 3 e 6 sofre alteração de conteúdo, ou seja, ficam falhos, no instante 27/07/2004 00:09:07.

Nodo 0

```
Testing: nodo [0] no tempo '27/07/2004 00:09:08' inicia testes.
  Test: nodo 0 tempo '27/07/2004 00:09:09' testa nodos (0 e 1). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 0 tempo '27/07/2004 00:09:09' testa nodos (0 e 2). DIAG: nodo 2 é ffree (conjunto 1).
  Test: nodo 0 tempo '27/07/2004 00:09:09' testa nodos (0 e 4). DIAG: nodo 4 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,ffree,1,4  >>
```

Nodo 1

```
Testing: nodo [1] no tempo '27/07/2004 00:09:08' inicia testes.
  Test: nodo 1 tempo '27/07/2004 00:09:08' testa nodos (1 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 1 tempo '27/07/2004 00:09:09' testa nodos (1 e 3). DIAG: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 1 tempo '27/07/2004 00:09:09' testa nodos (1 e 5). DIAG: nodo 5 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 1:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,FAULTY,2,3
          4,ffree,1,2  5,ffree,1,2  6,ffree,1,2  7,ffree,1,4  >>
```

Nodo 2

```
Testing: nodo [2] no tempo '27/07/2004 00:09:11' inicia testes (Rodada Teste: 22).
  Test: nodo 2 tempo '27/07/2004 00:09:11' testa nodos (2 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 2 tempo '27/07/2004 00:09:11' testa nodos (2 e 3). DIAG: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 2 tempo '27/07/2004 00:09:11' testa nodos (2 e 6). DIAG: nodo 6 é FAULTY (conjunto 3).
  Test: nodo 2 tempo '27/07/2004 00:09:11' testa nodos (2 e 7). DIAG: nodo 7 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 2:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,FAULTY,2,3
          4,ffree,1,2  5,ffree,1,2  6,FAULTY,3,3  7,ffree,1,4  >>
```

Nodo 7

```
Testing: nodo [7] no tempo '27/07/2004 00:09:12' inicia testes.
  Test: nodo 7 tempo '27/07/2004 00:09:12' testa nodos (7 e 3). DIAG: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 7 tempo '27/07/2004 00:09:12' testa nodos (7 e 5). DIAG: nodo 5 é ffree (conjunto 1).
  Test: nodo 7 tempo '27/07/2004 00:09:12' testa nodos (7 e 6). DIAG: nodo 6 é FAULTY (conjunto 3).
  Test: nodo 7 tempo '27/07/2004 00:09:12' testa nodos (7 e 2). DIAG: nodo 2 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 7:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,FAULTY,2,3
          4,ffree,1,2  5,ffree,1,2  6,FAULTY,3,3  7,ffree,1,4  >>
```

Nodo 4

```
Testing: nodo [4] no tempo '27/07/2004 00:09:14' inicia testes.
  Test: nodo 4 tempo '27/07/2004 00:09:14' testa nodos (4 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 4 tempo '27/07/2004 00:09:14' testa nodos (4 e 5). DIAG: nodo 5 é ffree (conjunto 1).
  Test: nodo 4 tempo '27/07/2004 00:09:14' testa nodos (4 e 6). DIAG: nodo 6 é FAULTY (conjunto 2).
    End of Testing Round: diagnostico do nodo 4:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,ffree,1,2
          4,ffree,1,2  5,ffree,1,2  6,FAULTY,2,3  7,ffree,1,4  >>
```

Nodo 5

```
Testing: nodo [5] no tempo '27/07/2004 00:09:14' inicia testes.
  Test: nodo 5 tempo '27/07/2004 00:09:15' testa nodos (5 e 1). DIAG: nodo 1 é ffree (conjunto 1).
    obtendo info: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 5 tempo '27/07/2004 00:09:15' testa nodos (5 e 4). DIAG: nodo 4 é ffree (conjunto 1).
  Test: nodo 5 tempo '27/07/2004 00:09:15' testa nodos (5 e 7). DIAG: nodo 7 é ffree (conjunto 1).
    obtendo info: nodo 6 é FAULTY (conjunto 3).
    End of Testing Round: diagnostico do nodo 5:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,FAULTY,2,3
          4,ffree,1,2  5,ffree,1,2  6,FAULTY,3,3  7,ffree,1,4  >>
```

Nodo 1

```
Testing: nodo [1] no tempo '27/07/2004 00:09:19' inicia testes.
  Test: nodo 1 tempo '27/07/2004 00:09:19' testa nodos (1 e 0). DIAG: nodo 0 é ffree (conjunto 1).
  Test: nodo 1 tempo '27/07/2004 00:09:19' testa nodos (1 e 3). DIAG: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 1 tempo '27/07/2004 00:09:19' testa nodos (1 e 5). DIAG: nodo 5 é ffree (conjunto 1).
    obtendo info: nodo 6 é FAULTY (conjunto 3).
    End of Testing Round: diagnostico do nodo 1:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,FAULTY,2,3
          4,ffree,1,2  5,ffree,1,2  6,FAULTY,3,3  7,ffree,1,4  >>
```

Nodo 0

```
Testing: nodo [0] no tempo '27/07/2004 00:09:19' inicia testes.
  Test: nodo 0 tempo '27/07/2004 00:09:19' testa nodos (0 e 1). DIAG: nodo 1 é ffree (conjunto 1).
  Test: nodo 0 tempo '27/07/2004 00:09:19' testa nodos (0 e 2). DIAG: nodo 2 é ffree (conjunto 1).
    obtendo info: nodo 3 é FAULTY (conjunto 2).
    obtendo info: nodo 6 é FAULTY (conjunto 3).
  Test: nodo 0 tempo '27/07/2004 00:09:20' testa nodos (0 e 4). DIAG: nodo 4 é ffree (conjunto 1).
    End of Testing Round: diagnostico do nodo 0:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,FAULTY,2,3
          4,ffree,1,2  5,ffree,1,2  6,FAULTY,3,3  7,ffree,1,4  >>
```

Nodo 4

```
Testing: nodo [4] no tempo '27/07/2004 00:09:25' inicia testes.
  Test: nodo 4 tempo '27/07/2004 00:09:25' testa nodos (4 e 0). DIAG: nodo 0 é ffree (conjunto 1).
    obtendo info: nodo 3 é FAULTY (conjunto 2).
  Test: nodo 4 tempo '27/07/2004 00:09:26' testa nodos (4 e 5). DIAG: nodo 5 é ffree (conjunto 1).
  Test: nodo 4 tempo '27/07/2004 00:09:26' testa nodos (4 e 6). DIAG: nodo 6 é FAULTY (conjunto 3).
    End of Testing Round: diagnostico do nodo 4:
      <<  0,ffree,1,0  1,ffree,1,2  2,ffree,1,2  3,FAULTY,2,3
          4,ffree,1,2  5,ffree,1,2  6,FAULTY,3,3  7,ffree,1,4  >>
```

Neste experimento, o primeiro nodo a realizar testes após o evento é o nodo 0 que inicia sua rodada de testes no instante 27/07/2004 00:09:08. Em seguida os nodos 1, 7, 2 e 4 realizam suas próximas rodadas de testes. Como estes nodos têm como filhos ao menos um dos dois nodos que sofreram o evento, estes nodos detectam novas informações sobre o estado dos nodos 3 e 6. Os nodos 2 e 7 já detectam os dois eventos ocorridos, mas os nodos 1 e 4 só detectam um evento cada, pois estes dois nodos só possuem respectivamente um dos nodos 3 e 6 como filhos.

Os próximos testes dos nodos 5 e 0 que são os nodos mais distantes dos eventos, acontecem respectivamente nos tempos 27/07/2004 00:09:14 e 27/07/2004 00:09:19. Neste momento, estes dois nodos obtêm informação sobre os dois eventos através de testes a nodos que já possuíam as informações. O nodo 5 testa os nodos 1 e 7 que repassam informação respectivamente sobre os nodos 3 e 6; e, o nodo 0 testa o nodo 2 que repassa as informações dos eventos ocorridos nos nodos 3 e 6. Assim, só resta aos nodos 1 e 4 detectarem o evento ocorrido respectivamente nos nodos 6 e 3, que o fazem através de testes aos nodos 3 e 0 respectivamente.

Capítulo 6

Conclusão

Este trabalho apresentou um novo modelo genérico de diagnóstico hierárquico adaptativo distribuído e baseado em comparações, e apresentou um novo algoritmo, *Hi-Dif*, baseado neste modelo, para o diagnóstico distribuído de alterações em sistemas com conteúdo replicado em uma rede como, por exemplo, a Internet. No modelo apresentado, os nodos do sistema podem estar falhos ou sem-falhas e um nodo falho pode tanto estar com falha do tipo crash, como apenas estar respondendo de maneira incorreta às tarefas enviadas como teste.

No algoritmo *Hi-Dif*, um nodo sem-falha testa outro nodo do sistema para classificar seu estado. O algoritmo classifica os nodos do sistema em conjuntos de acordo com o resultado dos testes e um teste é realizado através do envio de uma tarefa para um par de nodos do sistema. Após o recebimento das duas saídas da execução destas tarefas, o testador compara estas saídas e, se a comparação indicar igualdade, os nodos são classificados no mesmo conjunto de nodos. Por outro lado, se a comparação das saídas indicar diferença, os nodos são classificados em conjuntos distintos, de acordo com o resultado da tarefa. Um dos conjuntos contém os nodos sem-falha do sistema.

Uma diferença do modelo apresentado para outros modelos propostos anteriormente, é que a seguinte asserção foi eliminada: a comparação, realizada por um nodo sem-falha, das saídas produzidas por dois nodos falhos sempre resulta em diferença. Por este motivo, o modelo apresentado permite o tratamento da situação onde dois nodos quaisquer estão invadidos e vandalizados com as mesmas alterações sobre o conteúdo replicado.

Este novo algoritmo permite o diagnóstico de vandalismo em servidores Web com dados replicados. Como o algoritmo classifica os nodos em conjuntos de acordo com o resultado das tarefas enviadas como teste, este novo algoritmo permite a identificação dos nodos que estão com conteúdo modificado e diferenciá-los, por exemplo, de outros nodos que venham a possuir outras modificações de conteúdo.

Quando um nodo sem-falha testa outro nodo sem-falha, ele recebe do nodo testado informações de diagnóstico de todo o cluster ao qual o nodo testado pertence. Neste algoritmo os clusters são conjuntos com $N/2$ nodos. Como no algoritmo *Hi-Dif* é possível que o nodo i receba informações de diagnóstico do nodo j através de dois ou mais nodos, é necessário garantir que o nodo i sempre obtenha a informação mais recente sobre os outros nodos do sistema. Para garantir isso, o algoritmo implementa *timestamps*, que são implementados através de contadores de troca de estados, que cada nodo possui para todos os nodos do sistema.

Foi demonstrado que o algoritmo *Hi-Dif* possui uma latência de $log_2 N$ rodadas de testes para um sistema de N nodos, ou seja, é capaz de diagnosticar qualquer evento

ocorrido em no máximo $log_2 N$ rodadas de testes. Uma rodada de teste é um intervalo de tempo onde todos os nodos sem-falha do sistema obtêm informação de diagnóstico sobre todos os nodos do sistema. O algoritmo também é (N–1)-diagnosticável e o número máximo de testes requeridos pelo algoritmo é de $O(N^2)$, no pior caso.

Experimentos através de simulações do algoritmo e experimentos através da implementação do algoritmo utilizada para o diagnóstico de nodos com conteúdo replicado na Web também foram realizados. Os resultados dos experimentos comprovam o número máximo de testes necessários; comprovam também que a latência do algoritmo é $log_2 N$; e, mostram também que, mesmo quando N–1 nodos ficam falhos ao mesmo tempo, o único nodo sem-falha do sistema é capaz de diagnosticar todos os eventos.

Trabalhos futuros incluem o estudo da aplicação do algoritmo em redes peer-to-peer, e a implementação de um pacote para que a instalação e configuração do algoritmo possam ser facilmente executadas.

Referências Bibliográficas

[1] NUA.COM. How Many Online, http://www.nua.ie/surveys/how_many_online /world.html, Acessado em 02/10/2003.

[2] CERT Coordination Center, http://www.cert.org, Acessado em 03/10/2003.

[3] E. P. Duarte Jr., G. Mansfield, S. Noguchi, M. Miyazaki, "Fault-Tolerant Network Management," *Proc. ISACC'94*, Monterrey, Mexico, 1994.

[4] D. Ingham, S. K. Shrivastava, F. Panzieri, "Constructing Dependable Web Services," *IEEE Internet Computing*, Vol 4, No. 1, pp 25-33, Jan/Fev 2000.

[5] ALDAS, Analytisches Labor Dr. Axel Schumann, http://www.aldas.de, Acessado em 05/10/2003.

[6] IDG Now! - Hackers brasileiros lideram ranking de invasões em 2001, http://idgnow.terra.com.br/idgnow/internet/2002/01/0009, Acessado em 27/09/2003.

[7] IDG Now! - Cresce o número de invasões a sites Linux, http://idgnow.terra.com.br/idgnow/internet/2002/07/0046, Acessado em 05/10/2003.

[8] D. Russel, G. T. Gangemi, *Computer Security Basics*. O'Reilly, 1992.

[9] S. Garfinkel, G. Spafford, e A. Schwartz, *Practical Unix & Internet Security*, O'Reilly, 3a. ed., Fev. 2003.

[10] B. Tan, S. Foo, S. C. Hui, “Monitoring Web Information Using PBD Technique,” *Proc. 2nd International Conference on Internet Computing (IC’2001)*, Las Vegas, USA, pp. 666-672, Jun. 2001.

[11] L. Liu, C. Pu, W. Tang, “WebCQ – Detecting and Delivering Information Changes on the Web,” *Proceedings of International Conference on Information and Knowledge Managemente (CIKM)*, pp. 512-519, Nov. 2000.

[12] F. Douglis, T. Ball, “Tracking and Viewing Changes on the Web,” *1996 USENIX Techinical Conference*, pp. 165-176, 1996.

[13] Url Minder, http://www.netmind.com/URL-minder/URL-minder.html. Acessado em 22/09/2003.

[14] B. Lu, S. C. Hui, Y. Zhang, “Personalized Information Monitoring Over the Web,” *1st International Conference on Information Technology & Applications (ICITA 2002)*, Nov. 2002.

[15] D. Faensen, L. Faulstich, H. Schweppe, A. Hinze, A. Steidinger, “Hermes – A Notification Service for Digital Libraries,” *ACM/IEEE JCDL’01*, pp. 373-380, Jun. 2001.

[16] T. Catarci, “Web-based Information Access,” *IEEE Proceedings of the 4th IECIS International Conference on Cooperative Information Systems (CoopIS)*, Edinburgo, Escócia, pp. 10-19, 1999.

[17] V. Boyapati, K. Chevrier, A. Finkel, N. Glance, T. Pierce, R. Stockton, C. Whitmer, “ChangeDetector™: A Site-Level Monitoring Tool for the WWW,” *International World Wide Web Conference*, Hawaii, USA, pp. 570-579, Mai. 2002.

[18] N. Glance, J.-L. Meunier, P. Bernard, D. Arregui, “Colaborative Document Monitoring,” *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, Boulder, CO, 2001.

[19] S.-J. Lim, Y.-K. Ng, “An Automated Change-detection Algorithm for HTML documents Based on Semantic Hierachies,” *Proceedings of the 17th International Conference on Data Engineering (ICDE’01)*, Heidelberg, Alemanha, pp. 303-312, Abr. 2001.

[20] S. Flesca, F. Furfaro, E. Masciari, “Monitoring Web Information Changes,” *International Conference on Information Technology: Coding and Computing (ITCC’01)*, Abr. 2001.

[21] G. Masson, D. Blough, G. Sullivan, “System Diagnosis,” in *Fault-Tolerant Computer System Design*, D.K. Pradhan, Prentice-Hall, 1996.

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

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

[24] R. P. Bianchini, R. Buskens, “An Adaptive Distributed System-Level Diagnosis Algorithm and Its Implementation,” *Proc. FTCS-21*, 1991.

[25] P. Jalote, *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[26] E. P. Duarte Jr., T. Nanya, “A Hierarquical Adaptive Distributed System-Level Diagnosis Algotithm,” *IEEE Transactions on Computers*, Vol. 47, No.1, pp. 34-45, Jan 1998.

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

[28] A. Brawerman, E. P. Duarte Jr., “A Synchronous Testing Strategy for Hierarchical Adaptive Distributed System-Level Diagnosis,” *Proc. IEEE LATW'00*, pp. 154-161, 2000.

[29] E. P. Duarte Jr., A. Brawerman, L. C. P. Albini, “An Algorithm for Distributed Hierarquical Diagnosis of Dynamic Fault and Repair Events,” *Proc. IEEE International Conference on Parallel and Distributed Systems 2000*, pp. 299-306, 2000.

[30] M. Malek, “A Comparison Connection Assignment for Diagnosis of Multiprocessor Systems,” *Proc. 7th International Symp. Computer Architecture*, pp. 31-36, 1980.

[31] 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.

[32] J. Maeng, M. Malek, “A Comparison Connection Assignment for Self-Diagnosis of Multiprocessor Systems,” *Digest 11th Internationall Symp. Fault Tolerant Computing*, pp. 173-175, 1981.

[33] 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.

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

[35] D. Wang, “Diagnosability of Hipercubes and Enhanced Hypercubes under the Comparison Diagnosis Model,” *IEEE Transactions on Computers*, Vol. 48, No. 12, pp. 1369-1374, 1999.

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

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

[38] N. F. Tzeng, S. Wei, “Enhanced Hypercubes,” *IEEE Transactions on Computers*, Vol. 40, No. 3, pp. 284-294, Mar. 1991.

[39] T. Araki, Y. Shibata, “Diagnosability of Butterfly Networks under the Comparison Approach,” *IEICE Trans. Fundamentals*, Vol E85-A, No. 5, Maio 2002.

[40] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.

[41] J. Fan, “Diagnosability of Crossed Cubes,” *IEEE Transactions on Computers*, Vol. 13, No. 10, pp. 1099-1104, Out. 2002.

[42] E. P. Duarte Jr., L. C. P. Albini, A. Brawerman, A. L. P. Guedes, “A Hierarchical Distributed Diagnosis Algorithm with Detours,” *Technical Report 002/2003*, Universidade Federal do Paraná, Dept. Informática http://www.inf.ufpr.br/info/techrep, 2003.

[43] L. C. P. Albini, E. P. Duarte Jr., “Generalized Distributed Comparison-Based System-Level Diagnosis,” *2nd IEEE Latin American Test Workshop*, pp. 285-290, Set. 2001.

[44] S. L. Hakimi, A. T. Amin, “Characterization of Connection Assignment of Diagnosable Systems,” *IEEE Transactions on Computers*, Vol. 23, pp. 86-88, 1974.

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

[46] V. Hadzilacos, S. Toueg, *Fault-Tolerant Broadcasts and Related Problems*, “Distributed Systems,” S. Mullender, ACM Press, Cap. 5, 1993.

[47] T. Araki, Y. Shibata, “Efficient Diagnosis on Butterfly Networks under the Comparison Approach,” *IEICE Trans. Fundamentals*, Vol E85-A, No. 4, Abr. 2002.

[48] K. Efe, “A Variation on de Hypercube with Lower Diameter,” *IEEE Transactions on Computers*, Vol. 40, No. 11, pp. 1312-1316, Nov. 1991.

[49] K. Efe, “The Crossed Cube Architecture for Parallel Computing,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, pp. 513-524, Set. 1992.

[50] K. Efe, P. K. Blackwell, W. Slough, T. Shiau, “Topological Properties of the Crossed Cubes Architecture,” *IEEE Transactions on Computers*, Vol. 44, No. 7, pp. 923-929, Jul. 1995.

[51] F. Harary, *Graph Theory*, Addison-Wesley Publishing Company, 1971.

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

[53] E. P. Duarte Jr., L. C. E. Bona, “A Dependable SNMP-based Tool for Distributed Network Management,” *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'2002), Dependable Computing and Communications (DCC) Symposium*, Washington D. C., USA, 2002.

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

[55] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Prentice Hall, 2a ed., 1988.

[56] The Linux Home Page at Linux Online, http://www.linux.org, Acessado em 23/07/2004.

[57] D. E. Comer, *Internetworking with TCP/IP – Principles, Protocolos, and Architectures*, Prentice Hall, 4a ed., Vol. 1, 1995.

[58] Intel Corporation, http://www.intel.com, Acessado em 23/07/2004.

[59] Guia do Hardware.Net – htto://www.kurumin.com.br, Acessado em 23/07/2004.

[60] Rivest R, “The MD5 Message-Diggest Algorithm,” *RFC 1321*, IETF, Abr. 1992