UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO

Dissertação de Mestrado

UM ALGORITMO PARA DIAGNÓSTICO DISTRIBUÍDO DE FALHAS EM REDES DE COMPUTADORES

por

Moisés Almeida Castelo Branco

Orientador: Prof. Dr. Antônio Mauro Oliveira
Co-orientador: Prof. Msc. José Everardo Bessa Maia

Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado em Ciência da Computação

Moisés Almeida Castelo Branco

Um Algoritmo para Diagnóstico Distribuído de Falhas em Redes de Computadores

Este trabalho será apresentado à Pós-Graduação em Ciência da Computação, do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Antônio Mauro Oliveira

Co-orientador: Prof. Msc. José Everardo Bessa Maia

AGRADECIMENTOS

Meus sinceros agradecimentos ao Prof. José Everardo Bessa Maia pela confiança, e inestimável ajuda, fundamentais para a conclusão deste trabalho.

Agradeço também ao Prof. Antônio Mauro Oliveira pela atenção e precioso tempo dedicados.

Agradeço em especial aos meus pais, Oswaldo e Clara, pelo grande apoio e incentivo.

Agradeço ainda a todos os colegas que contribuíram com críticas, sugestões ou simples palavras de estímulo.

Por fim, sou grato sobretudo a Deus pela paz que sempre esteve comigo neste desafio.

ABSTRACT

In this work, we consider fault diagnosis in systems constituted of autonomous units, assuming that any two neighbor units are capable of performing tests on each other. Workstation networks are a typical example of such systems. The fault model is *fail-stop*, in which all faults are permanent. We describe a distributed algorithm to diagnose faults in nodes (processors) of a network, which is also correct in cases of faults in communication links. According to our literature revision concerning the subject, the former algorithms either worked only with faults in the processors, or demanded the knowledge of the complete connection topology of network in each node, to treat the faults in links. The described algorithm is applicable to the networks with general topology and is suitable for workstation networks. A node executing the algorithm, doesn't need to know about the complete connection topology, but only which other nodes are its neighbors. We present formal proof of the correctness of the algorithm and the results obtained by simulation in three types of connection topology of the network: networks with minimal connected topology, networks with full connected topology and networks with general topology. A practical implementation of the algorithm, in Java language, is also presented.

RESUMO

Redes de computadores apresentam muitas vantagens e têm se tornado muito comuns e imprescindíveis na maioria das organizações atualmente. Nesta dissertação, descrevemos, implementamos, testamos e comparamos o desempenho de um algoritmo para diagnóstico distribuído de falhas nos processadores de uma rede, que também é correto no caso de falhas nos enlaces de comunicação. O modelo de falhas é *fail-stop*, no qual todas as falhas são permanentes. De acordo com nossa revisão da literatura sobre o assunto, os algoritmos anteriores ou só trabalham com falhas nos processadores ou exigem o conhecimento de toda a topologia de interligação da rede em cada nó para tratar as falhas nos enlaces. O algoritmo proposto se aplica a redes de topologia geral e um nó executando o algoritmo não necessita conhecer toda a topologia de interligação da rede, mas apenas que outros nós são seus vizinhos. Provamos formalmente o corretismo do algoritmo, e apresentamos os resultados obtidos através de simulação considerando três tipos de topologia de interligação dos nós: redes de conexão mínima, redes completamente conectadas e redes de topologia geral. Uma implementação do algoritmo, feita em linguagem Java, também é apresentada.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LEMAS E TEOREMAS

CAPÍTULO 1

Introdução

1.1 Visão Geral

Redes de computadores têm recebido especial atenção como plataforma promissora para computação distribuída, cooperativa e tolerante a falhas. Durante as últimas três décadas, muito trabalho de pesquisa tem sido desenvolvido com o intuito de encontrar procedimentos eficientes para detectar e isolar circunstâncias anômalas (falhas) que comprometam o funcionamento destes sistemas. Em particular, se a disponibilidade da rede é essencial, como, por exemplo, em sistemas de tempo real e de controle de tráfego aéreo, é desejável que a presença de falhas não prejudique a própria tarefa de identificá-las.

O contínuo crescimento em número e diversidade dos equipamentos que compõem as redes tem tornado a atividade de localização das falhas cada vez mais complexa, sobretudo, se considerarmos o envolvimento de múltiplos fornecedores. A diversidade de soluções proprietárias para o monitoramento e isolamento de problemas, num ambiente heterogêneo, dificulta substancialmente um diagnóstico eficaz, levando em conta o fato de que, em geral, estas soluções não interagem umas com as outras.

Considerando a importância das redes na execução dos procedimentos operacionais de uma organização, os custos de um problema que cause uma falta de atendimento, ou uma queda inaceitável de sua qualidade, podem ser muito altos. Por outro lado, o diagnóstico incorreto de um problema ou ainda a tentativa de retomar o uso de um equipamento falho, antes que uma reparação tenha sido efetuada, elevam os custos e o tempo de paralisação dos serviços, aumentando a frustração dos usuários envolvidos.

Todos estes fatores têm causado uma crescente necessidade por ferramentas automáticas que se integrem ao próprio funcionamento da rede, e que possam efetivamente abstrair-se de detalhes inerentes a soluções específicas, oferecendo uma visão homogênea dos recursos e equipamentos. Tais ferramentas devem ser capazes de localizar com eficiência e uniformidade os problemas da rede, independente de seu tamanho ou heterogeneidade de seus componentes.

Nesta dissertação, trataremos o problema de detecção e localização de falhas em *unidades de processamento* (estações), interligadas através de uma rede de comunicação de topologia geral, que pode ser composta de enlaces ponto a ponto, canais de comunicação baseados em difusão (*broadcast*), ou uma combinação arbitrária destes. Em especial, nosso trabalho também lida de forma eficaz com a possibilidade de os enlaces de comunicação não serem confiáveis, isto é, também serem susceptíveis a falhas.

Apresentamos um algoritmo para *diagnóstico distribuído* de falhas em redes de computadores, que é tolerante à existência de múltiplas falhas tanto nas estações quanto nos enlaces de comunicação da rede. Faremos uma exposição da sua motivação e aplicabilidade, lançando mão de algumas definições que conceituam o ambiente ao qual ele se aplica. Em seguida, descrevemos detalhadamente o seu funcionamento e estruturas de dados.

Provamos formalmente que o algoritmo é correto, ou seja, garante a identificação precisa dos estados de funcionamento das estações da rede, dentro de condições que serão apresentadas ao longo do trabalho. Simulações de sua execução e análise de seu desempenho em redes de conexão mínima, completamente conectada e de topologia geral são feitas utilizando uma biblioteca para simulações. Finalmente, apresentamos também uma implementação do algoritmo, feita em linguagem *Java*.

1.2 Gerenciamento Aberto de Redes de Computadores[1]

O gerenciamento aberto de redes de computadores tem assumido, nos últimos anos, um papel preponderante dentro do cenário heterogêneo de tecnologias e soluções proprietárias para monitoramento e controle de suas funcionalidades.

Além de originar vários trabalhos de pesquisa, esta tarefa tem também despertado a atenção dos principais órgãos internacionais de normalização. A proposição de padrões e especificações que favoreçam a solução do problema e o domínio de sua complexidade tem sido alvo de significativos esforços. Se considerarmos a diversidade de tecnologias recém surgidas para a área de redes de computadores, e ainda os seus custos atraentes, concluímos que o fornecimento de um sistema aberto de gerenciamento que seja realmente efetivo, e garanta um comportamento previsível dentro de parâmetros de qualidade ideais, é, de fato, uma tarefa crítica.

A abrangência das atividades de um sistema aberto de gerenciamento de redes é bastante considerável. Usualmente, para uma melhor concepção de suas macro funções, adota-se classificá-las em cinco áreas principais, são elas: [STA 93]

- Gerenciamento de Falhas;
- Gerenciamento de Configuração;
- Gerenciamento de Contabilização;
- Gerenciamento de Desempenho;
- Gerenciamento de Segurança.

O gerenciamento de falhas é uma das áreas mais importantes de gerenciamento de redes. Essencialmente, o principal objetivo é garantir a identificação de comportamentos anômalos na rede, independente de sua complexidade e número de componentes.

[1] Usaremos esta terminologia para nos referirmos aos sistemas de gerenciamento de redes não proprietários, isto é, aos sistemas de gerenciamento independentes de qualquer fornecedor específico.

Diversos fatores devem ser considerados, dentre os quais, garantir a identificação precisa de componentes falhos, e também a provisão de mecanismos para isolá-los da porção restante da rede, de tal forma que ela possa continuar em funcionamento sem nenhuma interferência. Estas ações fazem parte dos procedimentos adotados com o objetivo de que a qualidade de serviço acertada com os usuários seja mantida dentro de padrões aceitáveis.

O gerenciamento de configuração é também uma área de muita importância e bastante desenvolvida. O seu objetivo principal é garantir a possibilidade de modificações com pequeno impacto no ambiente da rede, permitindo, por exemplo, modificar parâmetros de configuração, acrescentar ou remover componentes, e alterar a topologia de comunicação que os interliga.

O gerenciamento de contabilização preocupa-se, fundamentalmente, com a análise e registro de uso dos recursos da rede, sejam eles físicos ou lógicos. Estas atividades são de fundamental importância para garantir que não há usuários extrapolando a utilização dos recursos, ou ainda, utilizando-os de uma forma inadequada. Mensurações de uso tanto da quantidade quanto da capacidade dos recursos são responsabilidades desta área de gerenciamento.

O gerenciamento de desempenho é muitas vezes confundido com o gerenciamento de falhas. Muitos sistemas tendem a confundir desempenho com disponibilidade. Entretanto, há vários aspectos inerentes ao desempenho que não caracterizam precisamente uma falha, mas podem fazer com que a qualidade de serviços acertados com os usuários não seja satisfatória. Um tráfego excessivo em um canal de comunicação pode causar uma queda na qualidade de alguns serviços, entretanto, isto não caracteriza essencialmente uma falha. O objetivo do gerenciamento de desempenho é a detecção e correção deste tipo de circunstância que ponha em risco a qualidade de atendimento dos serviços.

O gerenciamento de segurança preocupa-se, essencialmente, com o controle de acesso aos recursos da rede. Nos últimos anos, esta área tem alcançado muita evolução, principalmente com a popularização do comércio eletrônico e a evolução das técnicas para codificação e criptografia de dados. Os mecanismos de acesso a informações

sigilosas, que empregam estratégias com uso de senhas e de chaves de criptografia, são atividades relacionadas a esta área de gerenciamento.

1.3 Gerenciamento de Falhas em Redes de Computadores

De um modo geral, os procedimentos para tratar a ocorrência de falhas num ambiente de gerenciamento aberto são agrupados nas seguintes categorias: [UDU 96]

- Detecção e geração de relatórios de falhas: Estes procedimentos incluem a propagação de *alarmes* e *eventos* para notificar os problemas e também a manutenção de históricos de suas ocorrências para auxiliar soluções futuras. Procedimentos para correlação e filtragem dos alarmes e eventos são também importantes, já que eles podem antecipar a ocorrência de problemas futuros ou iminentes e ainda decidir se os alarmes e eventos são realmente significativos para descobrir as causas do problema ou apenas efeitos colaterais.
- Diagnóstico de falhas: Dentre estes procedimentos incluem-se realização de testes e localização do problema.
- Correção de falhas: Estes procedimentos envolvem mecanismos manuais ou automáticos para solucionar um problema encontrado.
- Mapeamento de falhas: Estes procedimentos são responsáveis por mapear os problemas durante os seus ciclos de vida. Um mapeamento é usado para guardar detalhes de um problema desde a sua origem até que ele tenha sido resolvido, como, por exemplo, data e hora em que ele ocorreu, descrições de sua fonte ou provável causador, seu estado atual, etc.

Embora as especificações sobre gerenciamento de falhas descrevam em detalhes muitos aspectos e requisitos para um gerenciamento efetivo, e ainda especifiquem a organização geral das funções de gerenciamento utilizadas, algumas questões importantes são deixadas em aberto, como, por exemplo, as questões relativas às suas possíveis formas de implementação.

Especificamente na área de falhas, e em paralelo às estratégias convencionais adotadas pelos sistemas de gerenciamento aberto, significativos avanços têm sido conseguidos nos últimos anos na área de algoritmos para *Diagnóstico de Falhas em Sistemas de Unidades Autônomas* e, particularmente, em algoritmos para *Diagnóstico de Falhas em Redes de Computadores*. Estes algoritmos não se utilizam, à princípio, dos conceitos convencionais de gerenciamento de falhas adotados pelo sistemas de gerenciamento aberto. [DUA 98] Em contrapartida, os modelos convencionais de gerenciamento também não indicam direcionamentos específicos para implementação de suas macro atividades, dentre elas, inclusive, o procedimento de diagnóstico, que é, em último caso, o responsável pela localização das falhas existentes numa rede.

Em nosso trabalho, consideramos que a proposta de um algoritmo para o diagnóstico de falhas em uma rede de computadores pode ser apropriada para a integração ao gerenciamento de falhas no contexto de um sistema de gerenciamento aberto. Aliado a isto, discutiremos ao longo do trabalho algumas deficiências do modelo convencional de gerenciamento de falhas, que podem ser superadas com uso do algoritmo.

1.4 Diagnóstico de Falhas em Sistemas de Unidades Autônomas

Um sistema de unidades autônomas é aquele em que a falha em uma de suas unidades não ocasiona a falha de outras, ou seja, o sistema se compõe de unidades que possuem total independência funcional umas das outras. Uma rede de estações de trabalho (*workstations*) é um exemplo típico de sistema com esta estrutura.

Na literatura técnica, convencionou-se chamar de *diagnóstico*[1] o procedimento computacional para identificação e localização de unidades falhas neste tipo de sistema.

[1] *Diagnosis*, na literatura técnica de língua inglesa. O termo *diagnose*, da língua portuguesa, parece mais natural como uma tradução, entretanto, não o adotaremos em nosso trabalho, visto que ele se aplica corretamente apenas à área médica.

Após a execução de um procedimento de diagnóstico e, portanto, identificadas as unidades falhas, o sistema pode, por exemplo, ser reconfigurado logicamente, removendo-as de sua operação, ou ainda, disparar mecanismos para a reparação destas unidades, se elas forem estritamente necessárias.

Dentre as abordagens para procedimentos de diagnóstico, uma delas, em especial, tem sido alvo de diversas contribuições de pesquisas nos últimos anos, e é denominada na literatura técnica de língua inglesa por *System-Level Diagnosis*. A construção "Diagnóstico em Nível de Sistema", entretanto, não é bem aceita na norma oficial da língua portuguesa. Isto nos motivou a adotar, alternativamente no decorrer de nossa redação, o termo *Diagnóstico Automático de Sistema* para nos referirmos a esta linha de pesquisa, acreditando que ele seja mais apropriado. A adoção do termo *automático* nos parece ideal para significar o fato de que o objetivo da pesquisa é que, dada uma situação de falha, pelo menos uma unidade normal remanescente seja capaz de realizar um diagnóstico correto, com base em informações recebidas automaticamente sobre os estados de funcionamento das demais unidades, usando, por exemplo, uma rede de comunicação para o tráfego destas informações.

A abordagem de *Diagnóstico Automático de Sistema* já acumula uma fundamentação teórica sólida e também proporciona um *framework* prático, onde as falhas ao longo de uma rede possam ser *diagnosticadas* através de um modelo no qual as estações realizam testes umas nas outras.

Fundamentados nos princípios desta abordagem, consideramos neste trabalho o diagnóstico de falhas em estações de uma rede de computadores, assumindo que quaisquer duas estações vizinhas são capazes de realizar *testes* uma na outra.

1.5 Diagnóstico de Falhas em Redes de Computadores

De um modo geral, podemos representar uma rede de computadores por um *grafo conexo*, onde os processadores formam os vértices, e as ligações entre os processadores formam as arestas do grafo. Daqui em diante, chamaremos os processadores por *nós*, e as ligações entre eles por *enlaces de comunicação* ou, simplesmente, *enlaces*.

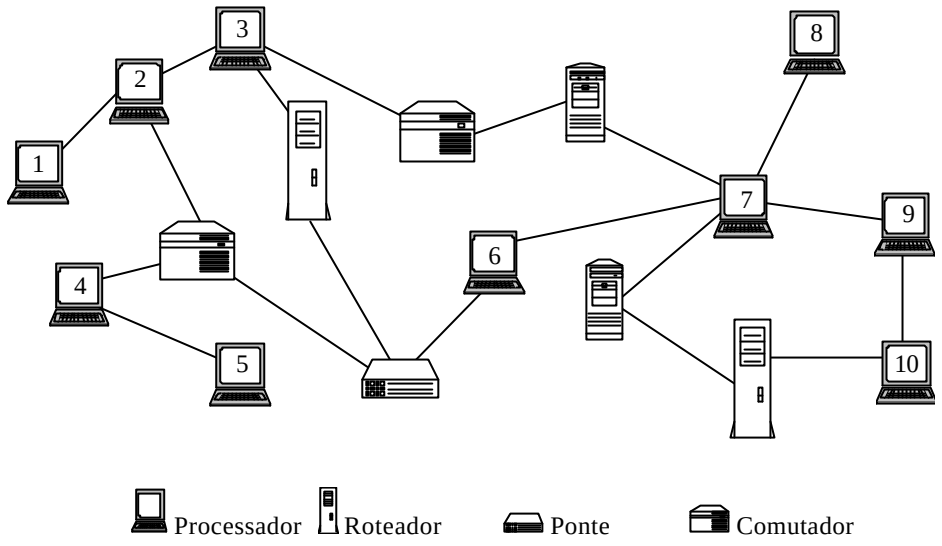


Figura 1.1 - Um exemplo de rede

A Figura 1.1 ilustra um exemplo de rede com dez nós conectados através de uma topologia de enlaces ponto a ponto[1]. Observando a Figura 1.1, podemos perceber que os enlaces de comunicação entre os nós podem apresentar diversas configurações - desde simples cabos físicos até o uso de equipamentos intermediários, dentre eles, por exemplo, pontes, roteadores ou comutadores[2]. A Figura 1.2 mostra um grafo representativo para este exemplo. Supondo que toda a rede está funcionando normalmente, e desconsiderando possíveis restrições de segurança, qualquer nó pode comunicar-se ou *atingir* todos os demais.

[1] Embora seja mais natural visualizarmos a representação em grafos para uma rede ponto a ponto, ela também se aplica a redes baseadas em difusão (ou *broadcast*) de mensagens (ex. *Ethernet*), neste caso, a representação seria um *grafo completo*.

[2] Este fato torna mais sensato considerar a possibilidade de falhas nos enlaces de comunicação.

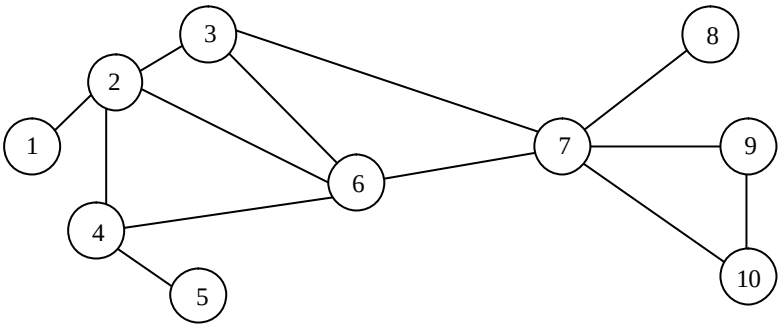


Figura 1.2 - Representação de uma rede através de um grafo

De um modo geral, qualquer sistema composto de múltiplas unidades, como, por exemplo, uma rede de computadores, pode estar sujeito a falhas em algumas ou todas elas. *Uma falha é uma disfunção no comportamento lógico normal esperado para uma unidade.* [TAN 95]

As falhas se classificam em três tipos: *transitórias, intermitentes ou permanentes*. *Falhas transitórias* são aquelas que ocorrem rapidamente (ou uma única vez) e depois desaparecem. Se a operação sobre uma unidade com este tipo de falha é repetida, a falha já terá desaparecido na maioria das vezes. *Falhas intermitentes* são aquelas que ocorrem raramente, ou se repetem a intervalos indefinidos, tal que as condições para a sua manifestação não podem ser reproduzidas. Falhas deste tipo são difíceis de diagnosticar. Uma *falha permanente* é aquela que se manifesta *continuamente*, causando que uma unidade sempre exiba um comportamento diferente do seu comportamento lógico normal, enquanto está falha.

O comportamento de uma unidade falha, por sua vez, se divide em dois tipos conforme as falhas sejam *Bizantinas*[1] ou *fail-stop*[2] (também chamada de *fail-silent*).

[1] *Byzantine faults*, na literatura técnica de língua inglesa.

[2] Usaremos a expressão original em inglês no decorrer do texto.

Uma unidade com uma falha *Bizantina* é aquela que continua em operação enquanto está falha, realizando processamentos incorretos e enganosos, podendo dar a impressão de que está funcionando normalmente, quando na verdade não está. Isto agrega uma complexidade substancial à tarefa de localização das unidades falhas.

Uma unidade com uma falha *fail-stop* é aquela que simplesmente cessa sua operação enquanto está falha, não sendo capaz de responder a entradas de dados ou ainda produzir resultados espúrios.

Nesta dissertação trataremos o problema de detectar e localizar *falhas permanentes* do tipo *fail-stop* em redes de computadores.

Numa rede de computadores convencional, há várias possibilidades de *falhas permanentes* do tipo *fail-stop*, dentre elas: falha em um único nó; falha em um único enlace de comunicação; ou ainda uma combinação arbitrária destas duas. Uma falha com estas características numa rede é aquela que causa que um nó ou enlace exiba *continuamente um comportamento diferente do normal, cessando suas respectivas operações*. Uma estação de trabalho desligada, ou um cabo de comunicação partido, são exemplos de falhas permanentes do tipo *fail-stop*, em uma rede de computadores convencional.

Dada uma *situação de falha* qualquer, devem sempre ser considerados dois casos em particular:

a. O sistema inteiro permanece *conexo*, isto é, todos os nós normais (não falhos) existentes ainda são capazes de se comunicar uns com os outros;

b. O sistema está separado em *componentes conexos*, isto é, há subconjuntos de nós normais que não são capazes de comunicar-se uns com os outros.

A Figura 1.3 mostra uma situação de falha que ilustra o *caso a*. Os enlaces representados por linhas tracejadas estão falhos, e o mesmo ocorre com o nó hachurado.

A Figura 1.4 ilustra o *caso b.*, onde vemos a existência de três *componentes conexos*, após um *evento de falha*. Neste caso, a falha no nó hachurado impede que quaisquer dois nós pertencentes a componentes conexos distintos possam comunicar-se.

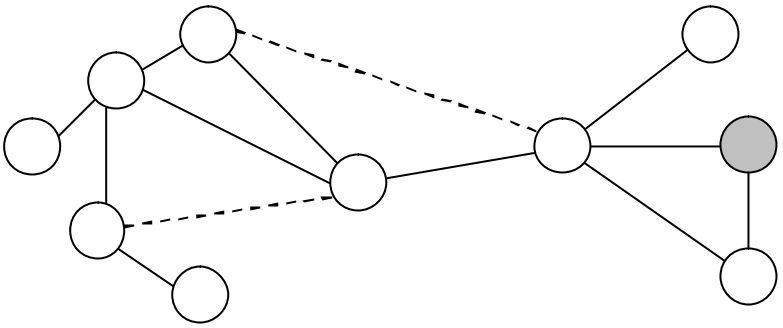


Figura 1.3 - Uma situação de falha em que o sistema permanece conexo

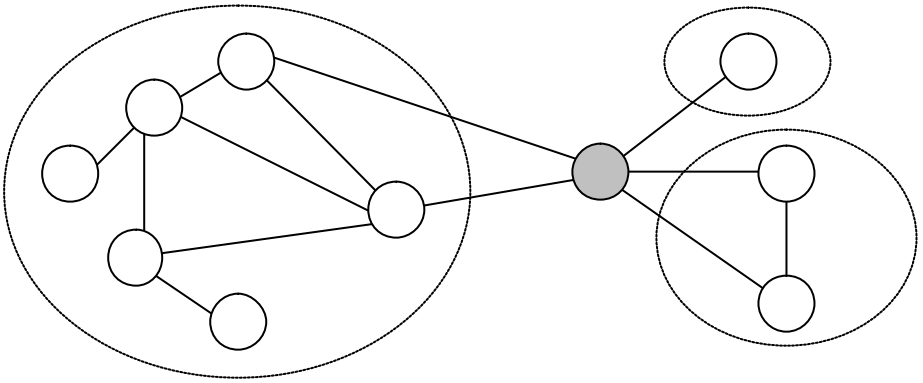


Figura 1.4 - Um sistema com três componentes conexos

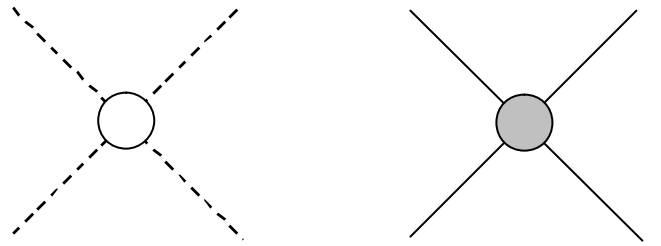
Por convenção, dois nós são ditos *vizinhos físicos* se estiverem ligados diretamente através de um enlace ponto a ponto, e, alternativamente, são ditos *vizinhos lógicos* se estiverem ligados através de um canal baseado em difusão (ou *broadcast*) de mensagens.

Numa rede de computadores, é usual que os nós realizem testes apenas em seus vizinhos (sejam eles físicos ou lógicos). Um nó pode realizar um teste em um vizinho, por exemplo, enviando uma mensagem para ele e iniciando um período de espera por uma mensagem de confirmação. A mensagem de confirmação é, geralmente, uma função da mensagem enviada. Isto tem como objetivo garantir que o nó sendo testado permaneceu normal durante todo o intervalo no qual o teste foi realizado. Caso o tempo máximo de espera por uma mensagem de confirmação tenha sido excedido (*time-out*), ou

uma mensagem de confirmação inválida tenha sido recebida, o nó que realizou o teste pode concluir que uma das três situações ocorre: o nó que foi testado está falho, o enlace que o conecta a ele está falho, ou ambos estão falhos.

Usualmente, cada nó também recebe através de seus vizinhos relatórios com os resultados de testes dos outros nós que ele não testa diretamente. Desta forma, um nó pode saber os estados de funcio namento de todos os demais.

Algumas configurações de falha, entretanto, não podem ser diagnosticadas corretamente usando apenas esta estratégia. Considere as duas configurações de falha ilustradas pela Figura 1.5. Na configuração de falha *A*, todos os enlaces de um nó estão falhos. Na configuração de falha *B*, o próprio nó está falho.



Configuração de Falha *A* Configuração de Falha *B*

Figura 1.5 - Configurações de falhas ambíguas

Utilizando apenas esta estratégia de testes, e propagação de mensagens entre nós vizinhos, é impossível para qualquer nó do sistema identificar precisamente qual destas duas situações de falha é a verdadeira.

Diante deste fato, um algoritmo de diagnóstico mais realista é aquele capaz de calcular a partir de um nó quais outros são *atingíveis* ou *inatingíveis*, independente de seus *estados de funcionamento* reais.

Na prática, três parâmetros de desempenho são utilizados para comparar algoritmos de diagnóstico: *número de testes por etapa*, *tráfego de mensagens* e *latência de diagnóstico*.

Uma *etapa de testes* é definida como o período de tempo em que cada nó tenha executado todos os testes a que é responsável. O número de testes por etapa é o somatório de todos os testes realizados por cada nó. O número de testes ideal (ótimo) é aquele em que cada enlace, ou cada nó, é testado uma única vez por etapa, dependendo se o algoritmo considera, ou não, falhas nos enlaces de comunicação.

O *tráfego de mensagens* é o principal dos parâmetros de comparação dos algoritmos de diagnóstico. Este parâmetro mede o *overhead* no tráfego na rede de comunicação, gerado pela execução do algoritmo de diagnóstico, enquanto propaga mensagens notificando eventos de falhas ou reparações. Em geral, este parâmetro é função da topologia de interligação dos nós.

A *latência de diagnóstico* é definida como sendo o tempo decorrido entre o acontecimento de uma falha ou reparação até o instante em que todos os nós normais remanescentes tenham obtido diagnóstico correto.

Um outro parâmetro importante para comparação de algoritmos de diagnóstico, que embora não seja um parâmetro de desempenho, também é usual, é a classificação *on-line* ou *off-line*. Um algoritmo de diagnóstico é dito *on-line*, se opera corretamente quando falhas e reparações ocorrem enquanto está sendo executado, e é dito *off-line* caso contrário.

Na literatura de *Diagnóstico Automático de Sistema*, há vários trabalhos propondo algoritmos para diagnosticar falhas em redes de computadores.

Bianchini e Buskens propuseram um algoritmo para diagnosticar falhas em processadores, não tratando a possibilidade de falhas nos enlaces de comunicação, chamado *Adaptive Distributed System-Level Diagnosis*. [BIA 92] Implementações e alguns refinamentos desta mesma abordagem foram ainda propostos por Duarte *et al.* [DUA 96,98] Estes algoritmos, entretanto, pressupõem que a rede de interligação entre os nós seja completamente conectada, seja física ou logicamente (ex. *Ethernet*).

Em [BAG 91], Bagchi e Hakimi introduziram um algoritmo para diagnóstico de falhas em processadores, conectados através de uma rede de topologia geral. Os nós normais formam uma topologia de testes baseada numa árvore, e as mensagens de diagnóstico são propagadas através da árvore. O algoritmo é ótimo em termos do tráfego

de mensagens, mas não funciona *on-line*, ou seja, um nó não pode falhar ou ser reparado durante a execução do algoritmo.

Em [STA 92], Stahl *et al.* apresentou e avaliou através de simulação o algoritmo *Adapt*, para diagnóstico de falhas em processadores, que trata a possibilidade de falhas nos enlaces de comunicação, e funciona *on-line*. A topologia de testes é um dígrafo minimamente conexo entre os nós normais. O principal problema deste algoritmo é que ele requer a troca de grandes quantidades de mensagens entre os nós, para reconstruir o dígrafo mínimo de testes, após cada evento de falha ou reparação.

Rangarajan *et al.* [RAN 95] introduziu um novo algoritmo para diagnóstico de falhas em nós conectados através de redes de topologia geral, que funciona *on-line*, e cada nó conhece apenas quem são os seus vizinhos. O algoritmo é ótimo no número de testes necessários por nó - cada nó é testado por um único vizinho, e apresenta ainda a menor *latência de diagnóstico*, usando uma estratégia de propagação de mensagens em paralelo. Apesar de ótimo no número de testes e na latência, o algoritmo não identifica falhas em uma configuração de falhas que os autores chamam de *jellyfish fault node configuration* e não trata a possibilidade de falhas nos enlaces de comunicação.

A principal deficiência dos algoritmos que não tratam falhas nos enlaces de comunicação é que um nó normal pode ser dado como falho, devido a uma falha em um de seus enlaces, mesmo que seja possível alcançá-lo por um caminho alternativo. Isto, portanto, limita consideravelmente a aplicação destes algoritmos.

Recentemente, Duarte *et al.* [DUA 98b] propôs um algoritmo para redes ponto a ponto de topologia geral, que funciona de forma *on-line*, e trata a possibilidade de falhas nos enlaces de comunicação. O algoritmo é ótimo no número de testes necessários, empregando uma estratégia de testes que os autores chamam de *two-way*, que supera o problema de detecção de falhas dada uma configuração *jellyfish*. O algoritmo também apresenta latência ótima, no entanto, para que funcione corretamente, é necessário que cada nó disponha do conhecimento de toda a topologia de interligação da rede. Isto pode ser um problema sério em redes espalhadas geograficamente, e que tenham sua topologia de interligação alterada com freqüência.

1.6 Contribuições desta Dissertação

Nesta dissertação desenvolvemos, implementamos, testamos e comparamos o desempenho de um algoritmo para diagnóstico distribuído de falhas em redes de computadores, que funciona de forma *on-line*, operando corretamente diante de falhas e reparações tanto de *nós* quanto de *enlaces*. Conforme nossa revisão bibliográfica, o estado da arte desta pesquisa é representado pelos trabalhos apresentados em [RAN 95] e [DUA 98b]. Como em [DUA 98b], o algoritmo proposto também considera falhas nos enlaces de comunicação, mas não necessita que cada nó conheça toda a topologia de interligação da rede - cada nó precisa saber apenas quais são os seus vizinhos. Entretanto, como em [RAN 95], o diagnóstico não é completo quando a situação de falha dividir o sistema em subconjuntos de nós normais que não atingem uns aos outros (*componentes conexos*). Em outras palavras, se o sistema está dividido em mais de um componente conexo, dois nós pertencentes a componentes conexos distintos podem manter estados desatualizados (incorretos) um do outro. Esta forma do algoritmo pode ser mais adequada em algumas situações particulares. Por exemplo, se a rede tem um alto grau de conectividade, a probabilidade da situação de falha dividir a rede em mais de um componente conexo é realmente pequena.

1.7 Organização da Dissertação

Organizaremos nossa dissertação como a seguir:

Capítulo 1 - Introdução

Neste capítulo, fazemos uma apresentação inicial do problema a ser tratado.

Capítulo 2 - Gerenciamento de Falhas no Contexto do Gerenciamento Aberto de Redes de Computadores

Neste capítulo, discutimos os conceitos de gerenciamento aberto de redes de computadores, e as estratégias usadas para o gerenciamento de falhas neste contexto. Faremos uma exposição da abordagem proposta pela ISO (*International Organization for Standardization*), discutindo em seguida algumas deficiências do modelo e a motivação para a proposta do algoritmo apresentado neste trabalho.

Capítulo 3 - Domínio do Problema e Pesquisas Relacionadas

Neste capítulo, apresentamos as principais pesquisas desenvolvidas na linha de *Diagnóstico Automático de Sistema*, que são relevantes ao nosso trabalho. Apresentamos alguns algoritmos para diagnóstico de falhas em redes de computadores, estudados na literatura técnica. Estes algoritmos serviram como fundamentação para a proposta do nosso algoritmo.

Capítulo 4 - Um Algoritmo para Diagnóstico Distribuído de Falhas em Redes de Computadores

Neste capítulo, apresentamos um algoritmo para diagnóstico distribuído de falhas em redes de computadores, baseado em outros algoritmos similares pesquisados na literatura técnica. As diferenças, vantagens e deficiências do nosso algoritmo, com relação aos demais, serão apresentadas em detalhes neste capítulo. São apresentadas ainda suas estruturas de dados, e estratégia de troca de mensagens. O capítulo contém também a prova formal de que o algoritmo é correto.

Capítulo 5 - Simulações e Desempenho

Neste capítulo, apresentamos diversos resultados da execução do algoritmo, obtidos através de simulação, usando a biblioteca SMPL. Apresentamos ainda algumas tabelas comparando os seus parâmetros de desempenho com os de outros algoritmos similares.

Capítulo 6 - Implementação

Neste capítulo, apresentamos uma implementação do nosso algoritmo, feita usando a linguagem *Java.*

Capítulo 7 - Conclusões

Neste capítulo, apresentamos as contribuições do nosso trabalho, os sumários de resultados, e ainda as perspectivas de trabalhos futuros.

Referências

Referências bibliográficas.

Apêndice A

Códigos fontes do simulador implementado em linguagem *C*, usando a biblioteca de rotinas para simulações de eventos discretos, SMPL.

Apêndice B

Códigos fontes do programa *NetInspector* - uma implementação do nosso algoritmo, usando a linguagem *Java.*

Apêndice C

Glossário de termos.

CAPÍTULO 2

Gerenciamento de Falhas no Contexto do Gerenciamento Aberto de Redes de Computadores

2.1 Introdução

Neste capítulo, faremos uma exposição sobre o gerenciamento de falhas no ambiente de gerenciamento aberto de redes OSI (*Open Systems Interconnection*) proposto pela ISO (*International Organization for Standardization*).

Na seção 2.2, apresentaremos os conceitos do sistema de gerenciamento OSI, e o modelo funcional empregado neste sistema.

Na seção 2.3, apresentaremos os mecanismos de gerenciamento de falhas adotados no sistema OSI.

Na seção 2.4, discutimos as deficiências dos mecanismos de gerenciamento de falhas empregados pelo sistema OSI.

2.2 Sistema de Gerenciamento Aberto de Redes OSI

O conjunto de especificações padronizadas para gerenciamento de redes de computadores, proposto pela ISO, é conhecido como sistema de gerenciamento aberto de redes OSI. Dentre estas especificações, incluem-se a definição dos serviços e protocolos de gerenciamento, a definição de um banco de dados, que mantém informações sobre os recursos gerenciados, e ainda as definições de diversos outros conceitos relacionados.

O primeiro trabalho realizado pela ISO, com relação à padronização da tarefa de gerenciamento de redes, está compilado no documento ISO7498-4, especificando as

linhas gerais do sistema de gerenciamento OSI. Este documento define que um sistema de gerenciamento aberto deve estar apto a satisfazer os seguintes requisitos:

- Suportar atividades que permitam planejar, organizar, supervisionar, controlar e contabilizar o uso dos serviços de comunicação.
- Responder eficientemente a mudanças em função de necessidade.
- Garantir um comportamento de comunicação previsível.
- Proporcionar a proteção das informações e a autenticação de fontes e destinos em uma transmissão de dados.

Ao refinarmos estes requisitos, chegaremos aos principais objetivos do sistema de gerenciamento aberto OSI, são eles:

- Garantir alta disponibilidade:
- Reduzir custos operacionais.
- Reduzir ou eliminar pontos críticos, distribuindo ou replicando a execução de atividades essenciais.
- Permitir flexibilidade de operação e integração, tornando transparente as mudanças ou incorporações de tecnologias diferentes.
- Aumentar a eficiência.
- Facilitar a utilização dos recursos
- Garantir segurança dos dados.

Todas as especificações que padronizam estas atividades de gerenciamento foram agrupadas pela ISO nas cinco categorias a seguir:

- *Visão geral e framework de gerenciamento OSI*: Inclui o ISO7498-4, o qual fornece uma introdução geral aos conceitos de gerenciamento, e o ISO10040 - um mapa para os conteúdos dos demais documentos.
- *CMIS/CMIP*: Define o Serviço de Informação de Gerenciamento Comum (CMIS - *Common Management Information Service*), o qual fornece os serviços de gerenciamento OSI para as aplicações de gerenciamento, e o

Protocolo de Informação de Gerenciamento Comum (CMIP - *Common Management Information Protocol*), o qual fornece os mecanismos de troca de informações que suportam o CMIS.

- *Funções de gerenciamento de sistemas*: Define as funções específicas para o gerenciamento OSI.
- *Modelo de informação de gerenciamento*: Define a base de informações de gerenciamento (MIB), que contém a representação de todos os objetos, dentro do ambiente OSI, sujeitos ao gerenciamento.
- *Gerenciamento de camadas*: Define as questões de gerenciamento, serviços, e funções específicas às camadas OSI.

Vejamos a seguir os principais conceitos empregados no ambiente de gerenciamento padronizado OSI.

2.2.1 Modelo de Gerenciamento OSI

O ambiente de gerenciamento OSI inclui os conceitos de *áreas funcionais*, *gerente, agente* e *objeto gerenciado*. Neste paradigma, um processo denominado gerente é responsável por coletar e enviar informações de gerenciamento a outros processos denominados agentes, num modelo funcional que aproxima-se da estratégia empregada na arquitetura cliente/servidor.

Os gerentes são responsáveis pelas ações de gerenciamento propriamente ditas, de acordo com uma das cinco áreas funcionais definidas pela ISO: gerenciamento de falhas, configuração, contabilização, desempenho e segurança. Os agentes são responsáveis por coletar dados operacionais e detectar eventos excepcionais em objetos gerenciados, os quais são abstrações de recursos reais e estão devidamente representados em uma base de dados denominada MIB (*Management Information Base*).

Gerentes e agentes trocam informações de gerenciamento através da rede usando um protocolo de comunicação padronizado. A Figura 2.1 ilustra o relacionamento entre gerente, agente e os objetos gerenciados.

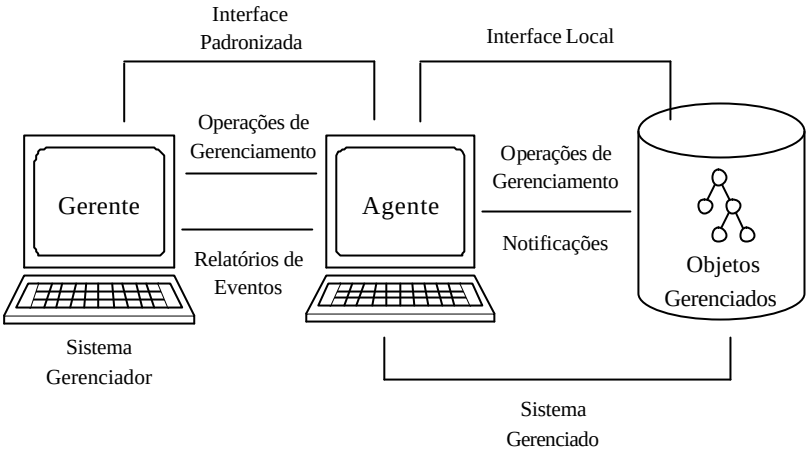


Figura 2.1 - Relacionamento entre gerente, agente e objetos gerenciados

Um gerente obtém dados atualizados sobre os objetos gerenciados e pode controlá-los. Para isso, transmite operações de gerenciamento aos agentes.

Um agente executa operações de gerenciamento sobre os objetos gerenciados, podendo ainda transmitir ao gerente as notificações emitidas por esses objetos.

Na concepção do gerenciamento OSI, um objeto gerenciado é a representação de um recurso que está sujeito ao gerenciamento. Os recursos podem tanto ser de *hardware* quanto de *software*, englobando desde dispositivos de comunicação até programas aplicativos. Os objetos gerenciados são definidos em termos de seus atributos, operações que podem ser efetuadas com eles, notificações que podem emitir para informar sobre a ocorrência de eventos especiais, e suas relações com outros objetos gerenciados.

O conjunto de todos os objetos gerenciados, e seus respectivos atributos dentro de um sistema, constituem a Base de Informação de Gerenciamento (MIB).

Um modelo arquitetural de gerenciamento OSI em cada componente participante do sistema é mostrado na Figura 2.2. Os elementos principais desta arquitetura são:

- Processo de Aplicação de Gerenciamento de Sistema (SMAP - *System Management Application Process*): Este é o processo local que é executado

dentro de um componente particular (estação de trabalho, processador de comunicação, roteador, etc.) responsável pela execução das funções de gerenciamento. Este processo tem acesso a todos os parâmetros do componente e exerce controle sobre ele, podendo coordenar-se com SMAPs de outros componentes.

- Entidade de Aplicação de Gerenciamento de Sistema (SMAE - *System Management Application Entity*): Esta entidade de aplicação é responsável pela troca de informações de gerenciamento através da rede de comunicação. O Protocolo de Informação de Gerenciamento Comum (CMIP - *Common Management Information Protocol*) é utilizado para este propósito.
- Entidade de Gerenciamento de Camada (LME - *Layer Management Entity*): Funcionalidade embutida em cada uma das camadas da arquitetura de rede OSI, que proporciona mecanismos de gerenciamento específicos para cada uma delas.
- Base de Informação de Gerenciamento (MIB - *Management Information Base*): O conjunto de todas as informações pertinentes ao sistema de gerenciamento.

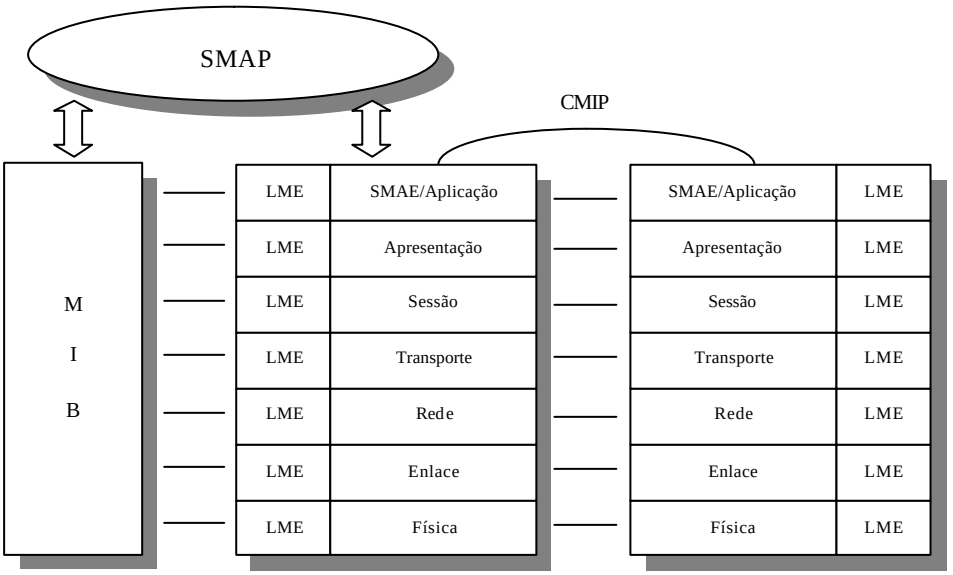


Figura 2.2 - Modelo de gerenciamento OSI

Um SMAP pode assumir o papel de gerente ou de agente, utilizando os serviços de comunicação da SMAE para trocar as informações e comandos gerenciamento. A comunicação é realizada utilizando os protocolos OSI. O serviço geral de gerenciamento OSI é o CMIS (*Common Management Information Service*). Outros serviços podem ainda ser utilizados, como, por exemplo o Serviço de Transferência, Acesso e Gerenciamento de Arquivos (FTAM - *File Transfer Access and Management*). O modo como os componentes de gerenciamento se inserem na camada de aplicação é mostrado pela Figura 2.3.

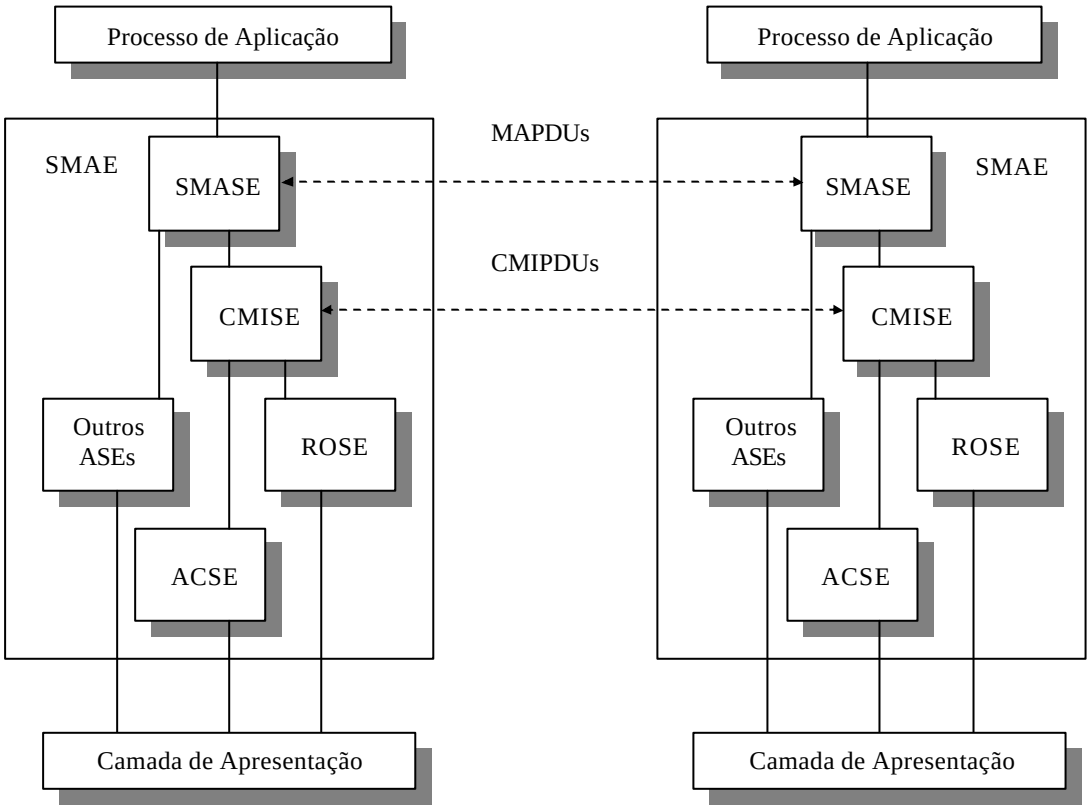


Figura 2.3 - Gerenciamento e a camada de aplicação

A Entidade de Aplicação de Gerenciamento de Sistemas (SMAE) consiste do Elemento de Serviço de Aplicação de Gerenciamento de Sistemas (SMASE - *Systems Management Application Service Element*), do Elemento de Serviço de Controle de Associação (ACSE - *Association Control Service Element*) e de outros Elementos de Serviço de Aplicação (ASEs - *Application Service Elements*), conforme descrito a seguir.

O SMASE define a semântica e a sintaxe abstrata da informação transferida nas Unidades de Dados do Protocolo de Aplicação de Gerenciamento (MAPDUs - *Management Application Protocol Data Units*). O SMASE especifica também a informação de gerenciamento a ser trocada entre SMAEs.

Os serviços de comunicação usados pelo SMASE podem ser prestados pelo Elemento de Serviço de Informação de Gerenciamento Comum (CMISE - *Common Management Information Service Element*) ou por outros ASEs, como o de Transferência, Acesso e Gerenciamento de Arquivos (FTAM - *File Transfer, Access and Management*) e o de Processamento de Transações (TP - *Transaction Processing*).

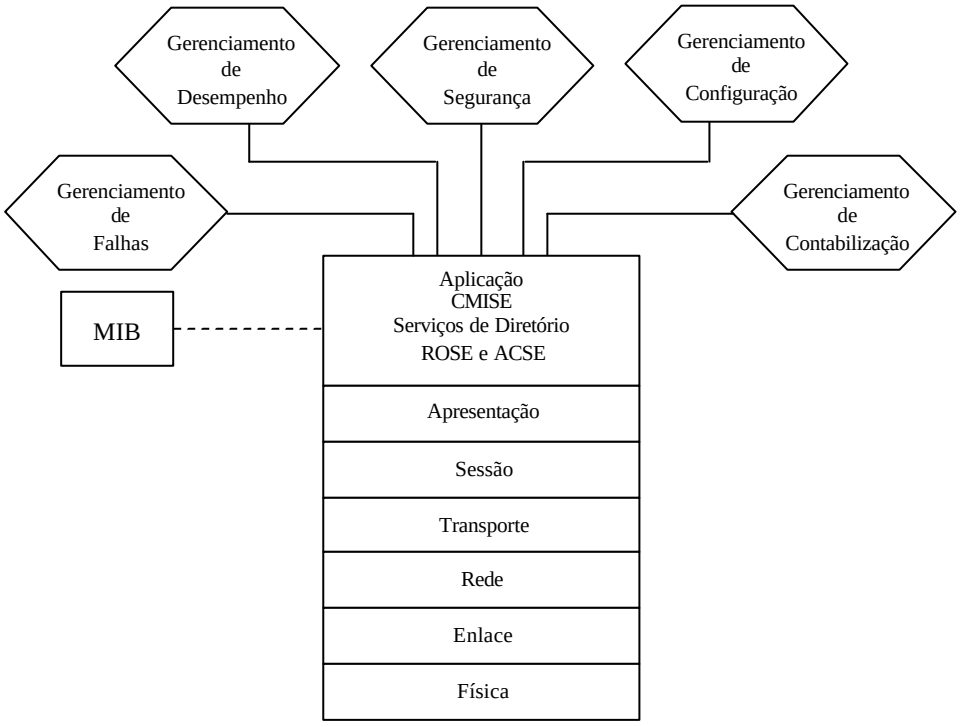


Figura 2.4 - Áreas funcionais de gerenciamento

O uso do CMISE implica na existência do Elemento de Serviço de Operações Remotas (ROSE - *Remote Operations Service Element*). O CMISE especifica o serviço e os procedimentos usados para a transferência das Unidades de Dados do Protocolo de Informação de Gerenciamento Comum (CMIPDUs - *Common Management Information Protocol Data Units*) e proporciona um meio de troca de informações para as operações de gerenciamento.

Dois SMAEs estabelecem uma associação acertando um contexto de aplicação que identifica o conhecimento inicial de gerenciamento compartilhado para aquela associação, incluindo os vários ASEs usados. A Figura 2.4 mostra o modelo de gerenciamento como suporte às áreas funcionais definidas pela ISO.

2.2.2 Conhecimentos de Gerenciamento

As informações que necessitam ser compartilhadas entre os SMAEs para fins de gerenciamento são denominadas de conhecimento de gerenciamento compartilhado (SMK – *Shared Management Knowledge*). Este conhecimento de gerenciamento inclui, principalmente, os seguintes elementos:

- O protocolo utilizado;
- As funções e unidades funcionais suportadas;
- As informações sobre os objetos gerenciados (por exemplo, classes, instâncias e identificação de objetos gerenciados e seus atributos);
- As restrições nas funções suportadas e relações entre estas funções e os objetos gerenciados.

A Figura 2.5 apresenta as visões do conhecimento de gerenciamento compartilhado. O conhecimento de gerenciamento pode ser estabelecido a qualquer momento, especificamente:

- Antes de a associação ser estabelecida;
- Durante a fase de estabelecimento da associação;

- Subseqüentemente, durante o tempo de vida da associação.

No estabelecimento da associação pode-se definir ou alterar o conhecimento de gerenciamento.

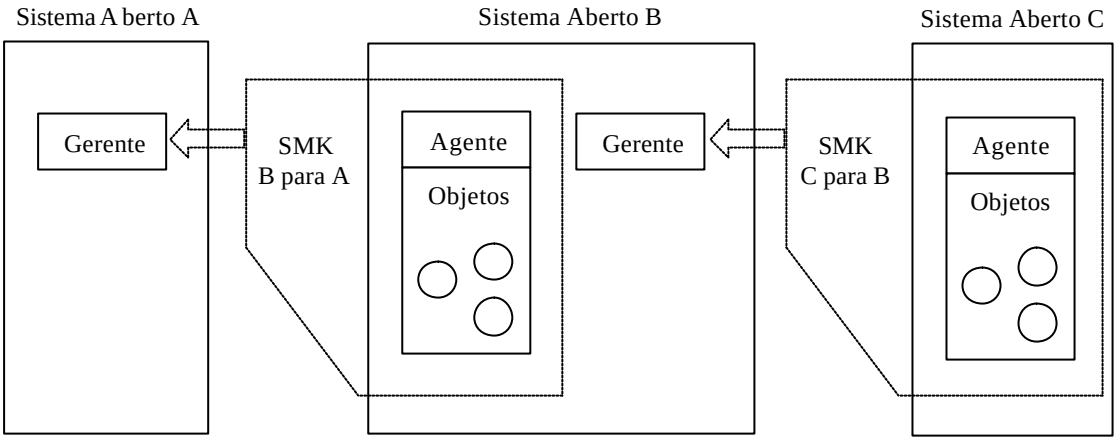


Figura 2.5 - Visões do conhecimento de gerenciamento compartilhado

2.2.3 Funções de Gerenciamento

As áreas funcionais de gerenciamento definidas no *framework* OSI descrevem as atividades gerais de responsabilidade do gerenciamento de redes. Para conveniência da padronização, estas atividades foram subdivididas pela ISO, nas normas ISO10164-1 a ISO10164-13, em *Funções de Gerenciamento de Sistemas* (SMFs - *Systems Management Functions*). Estas funções estão descritas a seguir:

1. Função de Gerenciamento de Objeto (OMF - *Object Management Function*): Permite a criação e remoção de objetos gerenciados e ainda a consulta e alteração de valores dos atributos destes objetos. Também especifica as notificações que podem ser emitidas por eles.
2. Função de Gerenciamento de Estado (STMF - *State Management Function*): Especifica um modelo para representação do estado de um objeto gerenciado.

3. Atributos para Representação de Relacionamento (ARR - *Attibutes for Representing Relationship*): Especifica um modelo para representar e controlar relacionamentos entre objetos gerenciados.

4. Função de Relatório de Alarme (ARF - *Alarm Report Function*): Permite a definição de alarmes para notificação de falhas ou situações críticas

5. Função de Gerenciamento de Relatório de Evento (ERMF - *Event Report Management Function*): Permite o controle e emissão de relatórios de eventos, incluindo as definições dos relatórios, os critérios usados para distribuí-los e a especificação de seus recipientes.

6. Função de Controle de *Log* (LCF - *Log Control Function*): Permite a criação de *Logs*, o armazenamento e recuperação de registros de *Logs*, e especifica os critérios para mantê-los.

7. Função de Relatório de Alarme de Segurança (SARF - *Security Alarm Reporting Function*): Permite a definição de alarmes relacionados à segurança e as notificações usadas para emiti-los.

8. Função de Registro para Auditoria de Segurança (SATF - *Security Audit Trail Function*): Especifica os tipos de relatórios de eventos que podem estar contidos em um histórico para fins de auditoria de segurança.

9. Objetos e Atributos para Controle de Acesso (OOAC - *Objects and Attibutes for Access Control*): Suporta o controle de acesso ao gerenciamento de informações e operações de gerenciamento.

10. Função de Medição de Contabilização (AMF - *Accounting Metering Function*): Fornece mecanismos para contabilização do uso dos recursos do sistema e um mecanismo para garantir limites para o uso dos recursos.

11. Função de Monitoração de Carga de Trabalho (WMF - *Workload Monitoring Function*): Suporta o monitoramento dos atributos dos objetos gerenciados relacionados ao desempenho de um recurso.

12. Função de Gerenciamento de Teste (TMF - *Test Management Function*): Suporta procedimentos para realização de testes e diagnóstico.

13. Função de Sumarização (SF - *Summarization Function*): Permite a definição de medidas estatísticas para aplicar aos valores de atributos dos objetos gerenciados e gerar relatórios com informações resumidas.

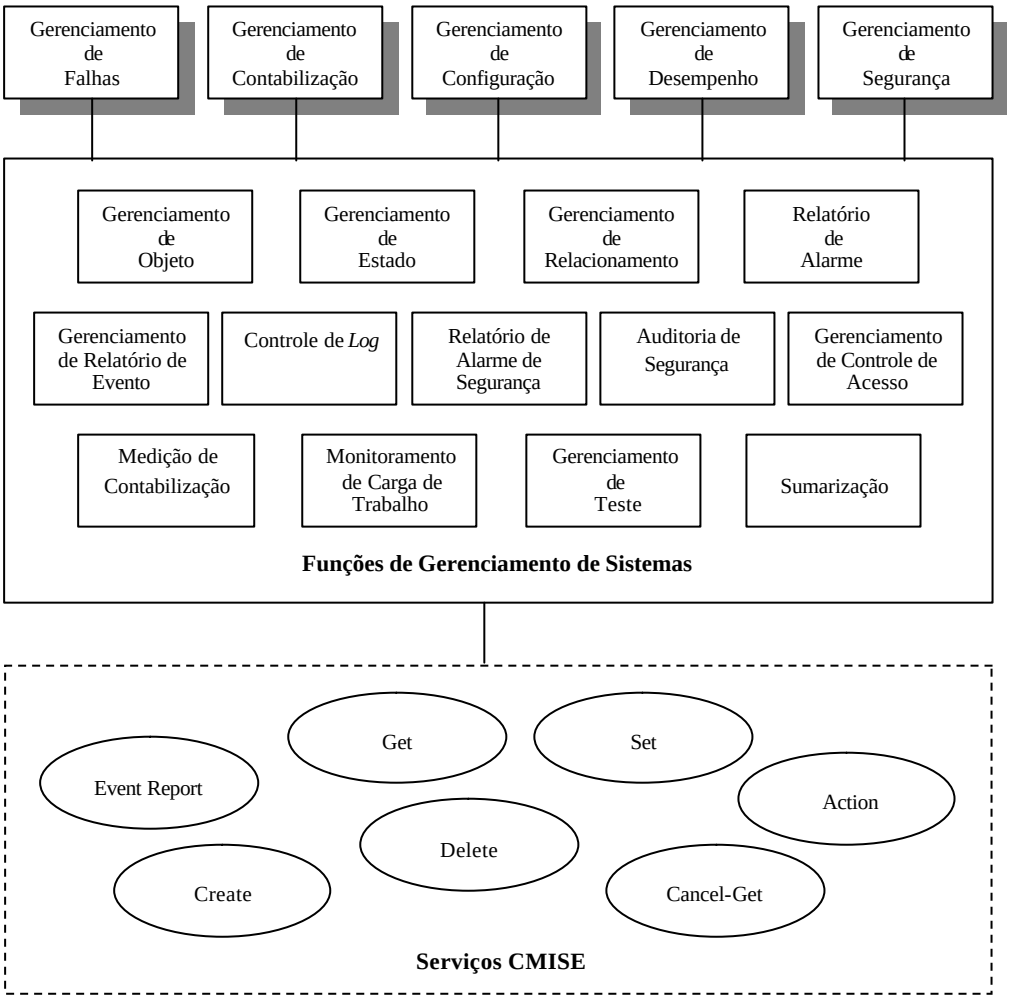


Figura 2.6 - Visão geral das áreas e funções de gerenciamento

Devido à superposição dos requisitos e necessidades dos usuários do sistema de gerenciamento, algumas destas funções podem ser usadas como suporte às atividades de mais de uma área funcional. A Figura 2.6 ilustra o relacionamento entre as áreas funcionais e as Funções de Gerenciamento de Sistemas (SMFs).

2.3 Gerenciamento de Falhas no Sistema OSI

O gerenciamento de falhas no ambiente de gerenciamento aberto preocupa-se, essencialmente, com a detecção e correção de situações anormais que impeçam o atendimento ou comprometam de forma inaceitável a qualidade dos serviços prestados aos usuários.

Neste sentido, o sistema OSI fornece quatro Funções de Gerenciamento de Sistemas (SMFs) que estão diretamente relacionadas ao gerenciamento de falhas, embora não necessitem ser utilizadas apenas para esta finalidade, são elas:

- Função de Relatório de Alarme (ARF - *Alarm Report Function*): Esta função está definida no documento ISO10164-4, e especifica um modelo para que informações notificando a detecção de circunstâncias críticas no sistema possam ser emitidas e tratadas pelo sistema de gerenciamento.
- Função de Gerenciamento de Relatório de Evento (ERMF - *Event Report Management Function*): Esta função está definida no documento ISO10164-5, e tem como objetivo definir um serviço para controlar a emissão de relatórios de eventos e os critérios para especificar os destinatários destes relatórios.
- Função de Controle de *Log* (LCF - *Log Control Function*): Esta função está definida no documento ISO10164-6, e tem como objetivo definir mecanismos para armazenar e recuperar históricos de informações (*Logs*) sobre o sistema de gerenciamento, como, por exemplo, dados sobre eventos que tenham ocorrido ou operações que tenham sido efetuadas nos objetos gerenciados.
- Função de Gerenciamento de Teste (TMF - *Test Management Function*): Esta função está definida no documento ISO10164-12, e tem como objetivo a definição de mecanismos para realização de testes sobre os recursos gerenciados.

Em grandes redes, onde há centenas ou milhares de equipamentos, torna-se bastante difícil isolar um problema quando ele ocorre. Quando se tem indícios de que há uma situação anormal, é importante identificar precisamente o que a está causando e

efetuar as medidas corretivas necessárias, como, por exemplo, reconfiguração da rede ou reparação do componente falho. Guardar informações detalhadas sobre os problemas e suas soluções em um arquivo de histórico (*Log*) é uma atividade usual para auxiliar a análise e solução de problemas futuros. Um problema pode ter várias causas, e é perfeitamente possível que um mesmo problema seja responsável pela geração de vários alarmes diferentes. Isto torna necessária a correlação dos alarmes, com o intuito de diferençar precisamente os que indicam de fato as razões do problema, daqueles que são apenas efeitos colaterais.

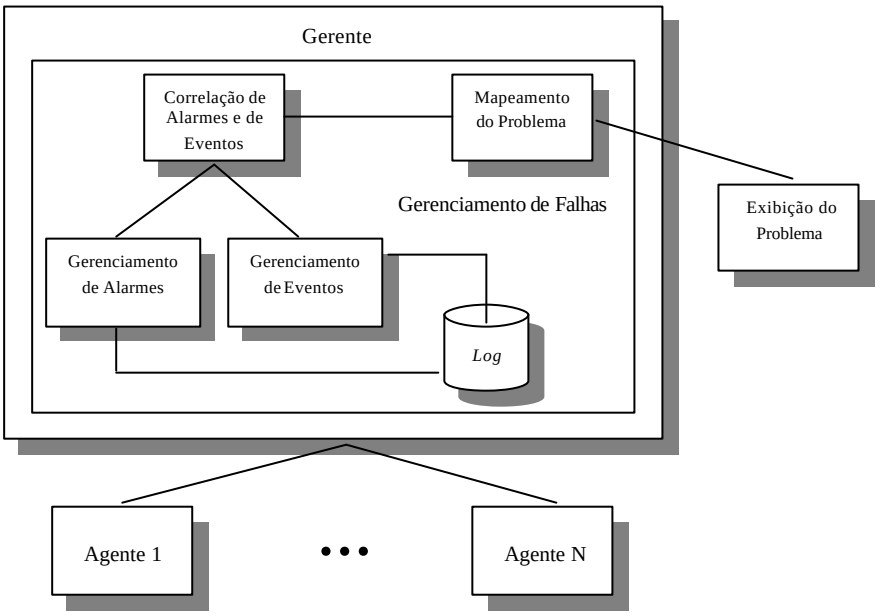


Figura 2.7 - Organização do gerenciamento de falhas no ambiente OSI

A Figura 2.7 ilustra a organização do gerenciamento de falhas dentro do sistema de gerenciamento OSI.

2.3.1 Função de Relatório de Alarme

Alarmes são notificações emitidas pelos objetos gerenciados, quando detectam-se no sistema as ocorrências de condições inusitadas ou anormais. Os alarmes podem ser gerados por mais de uma razão, e assim, para que as suas fontes sejam isoladas, eles

precisam ser correlacionados. O alarmes devem ser propagados de uma forma padronizada, e carregar informações suficientes para a identificação da natureza e fonte do problema, possibilitando uma ação correta para tratá-lo.

A Função de Relatório de Alarme (ARF - *Alarm Report Function*) foi definida com o objetivo de atender a esses requisitos, fornecendo os serviços e parâmetros necessários para a emissão e controle de alarmes.

Foram definidos os seguintes cinco tipos de alarmes:

- *Alarme de Comunicação*: Está relacionado com a transferência de informações de um ponto a outro dentro do sistema de gerenciamento.
- *Alarme de Qualidade de Serviço*: Está relacionado com a impossibilidade de atendimento ou queda na qualidade dos serviços acertados com os usuários.
- *Alarme de Processamento*: Está relacionado a falhas de *software* ou de processamento.
- *Alarme Ambiental*: Está relacionado às condições do ambiente em que estão alocados fisicamente os equipamentos que compõem o sistema de gerenciamento.

Cada alarme, por sua vez, é composto de uma série de atributos, que podem ser obrigatórios (O), condicionais (C), ou definidos pelo usuário (U). Os atributos que compõem um alarme são os seguintes:

1. Causas Prováveis (O): Indica a provável causa do alarme. A ISO define as seguintes causas prováveis para a emissão de um alarme:
    - *Perda de Sinal;*
    - *Erro de Framming;*
    - *Erro de Transmissão Local;*
    - *Erro de Transmissão Remota;*
    - *Erro de Estabelecimento de Chamada;*
    - *Sinal Degradado;*

- *Tempo de Resposta Excessivo;*
- *Tamanho da Fila Excedido;*
- *Largura da Banda Reduzida;*
- *Taxa de Retransmissão Excessiva;*
- *Limiar Atingido;*
- *Problema de Capacidade de Armazenamento;*
- *Incompatibilidade de Versões;*
- *Dados Danificados;*
- *Limite de Ciclos de CPU Excedido;*
- *Erro de Software;*
- *Fora da Área de Memória;*
- *Recurso não Disponível;*
- *Problema de Alimentação;*
- *Problema de Temporização;*
- *Problema do Cartão do Tronco;*
- *Problema do Cartão de Linha;*
- *Problema do Processador;*
- *Problema do Terminal;*
- *Problema do Dispositivo de Interface Externa;*
- *Problema do Conjunto de Dados;*
- *Problema do Multiplexador;*
- *Falha no Receptor;*
- *Falha no Transmissor;*
- *Detecção de Fumaça;*
- *Abertura da Porta do Compartimento;*

- *Temperatura do Ambiente Alta/Baixa;*
- *Umidade Alta/Baixa;*
- *Detecção de Intruso;*
- *Falha no Sistema de Refrigeração/Aquecimento;*
- *Falha no Sistema de Ventilação;*
- *Fogo;*
- *Inundação;*
- *Gás Tóxico;*
- *Pressão Alta/Baixa;*
- *Falha no Compressor de Ar;*
- *Falha na Bomba;*
- *Falha na Máquina;*
- *Problema no Fluído.*

2. Problemas Específicos (U): Fornece detalhes adicionais sobre as causas do problema.
3. Nível de Severidade (O): Especifica o grau de mau funcionamento ou degradação da qualidade do serviço oferecido ao usuário do sistema, com base no estado da capacidade de um determinado objeto gerenciado. Nas padronizações da ISO, são definidos seis níveis de severidade de um alarme, os quais estão descritos a seguir, em ordem crescente de gravidade:
    - *Removido*: Este nível de severidade indica que um alarme foi desconsiderado. Isto ocorre quando há alarmes do mesmo tipo, causa e problema, em que alguns, redundantes, são descartados.
    - *Indeterminado*: O nível de severidade é indeterminado quando não é possível estabelecer precisamente até que ponto o atendimento dos serviços será afetado.
    - *Alerta*: Este nível de severidade indica uma suspeita sobre a ocorrência de falhas antes que seus efeitos reais sejam percebidos. Devem ser realizados

procedimentos de diagnóstico sobre o recurso gerenciado, com o objetivo de corrigir um eventual problema.

- *Menor*: Este nível de severidade indica a existência de falhas comprometendo o funcionamento apropriado do sistema. Não há ainda uma degradação substancial do recurso gerenciado, mas ações corretivas devem ser tomadas para corrigir o problema.
- *Maior*: O nível de severidade é considerado maior quando há uma degradação sensível do funcionamento do recurso gerenciado. Neste caso, ações corretivas urgentes devem ser tomadas.
- *Crítico*: Este é o mais grave dos níveis de severidade, indicando uma total impossibilidade de funcionamento do recurso gerenciado. Ações corretivas urgentes devem ser tomadas para a reparação e retomada de funcionamento do recurso.

4. Estado de Cópia de Segurança (U): Indica se o objeto gerenciado que está emitindo o alarme possui ou não uma cópia de segurança (*backup*). Dependendo do nível de severidade do alarme, o sistema pode decidir utilizar esta cópia, se ela existir.
5. Objeto Cópia de Segurança (C): Está presente apenas se o Estado de Cópia de Segurança for verdadeiro, indicando qual o objeto que representa a cópia de segurança.
6. Indicação de Tendências (U): Esta informação é usada para indicar tendências demonstradas a partir do alarme atual com relação aos alarmes emitidos anteriormente, e que ainda não tenham sido removidos. Nas normas da ISO, são definidos os seguintes valores para este tipo de informação:
    - *Mais Severo*: O nível de severidade do alarme atual é maior do que os dos alarmes notificados anteriormente.
    - *Nenhuma Mudança*: O nível de severidade do alarme atual é o mesmo dos alarmes notificados anteriormente.
    - *Menos Severo*: Indica que existe pelo menos um alarme notificado anteriormente com um nível de severidade maior do que o do alarme atual.

7. Informação de Valor Limite (C): Esta informação é importante quando o alarme é resultado da extrapolação de um valor limite (*threshold*). Ela é definida através de quatro subparâmetros:
    - *Valor Limite Pré-Fixado*: Identificador do atributo cuja extrapolação do seu valor limite causou a notificação do alarme;
    - *Nível Limite*: Valor limite aceito para o atributo;
    - *Valor Observado*: Valor observado que ultrapassa o limite pré-estabelecido;
    - *Instante da Extrapolação*: Instante em que ocorreu a extrapolação do valor limite.
8. Identificador de Notificação (U): A identificação do alarme que pode ser usada para referências futuras por outros alarmes.
9. Notificações Correlacionadas (U): Um conjunto de Identificadores de Notificações com os quais considera-se que este alarme esteja relacionado.
10. Definições de Mudanças de Estado (U): Incluída quando há a transição do estado de um objeto associada com o alarme.
11. Atributos Monitorados (U): Identifica um ou mais atributos do objeto gerenciado, e seus respectivos valores, no momento em que o alarme foi emitido.
12. Sugestões para Ações de Reparação (U): Uma enumeração de possíveis ações que podem ser tomadas com o objetivo de solucionar o problema notificado pelo alarme.
13. Texto Adicional (U): Uma descrição informal do problema sendo notificado.
14. Informação Adicional (U): Informações adicionais do problema sendo notificado.

2.3.2 Função de Gerenciamento de Relatório de Evento

As notificações emitidas pelos objetos gerenciados devem ser seletivamente manipuladas, escolhendo algumas delas antes de enviá-las a um ou mais sistemas gerenciados. Para manipular o modo pelo qual as notificações são emitidas, dois objetos gerenciados, o *Discriminador* e o *Discriminador de Repasse de Eventos* foram definidos

pela ISO e estão descritos no documento ISO10165-2. Além destes dois objetos, a Função de Gerenciamento de Relatório de Evento (ERMF - *Event Report Management Function*) foi também definida e está descrita no documento ISO10164-5.

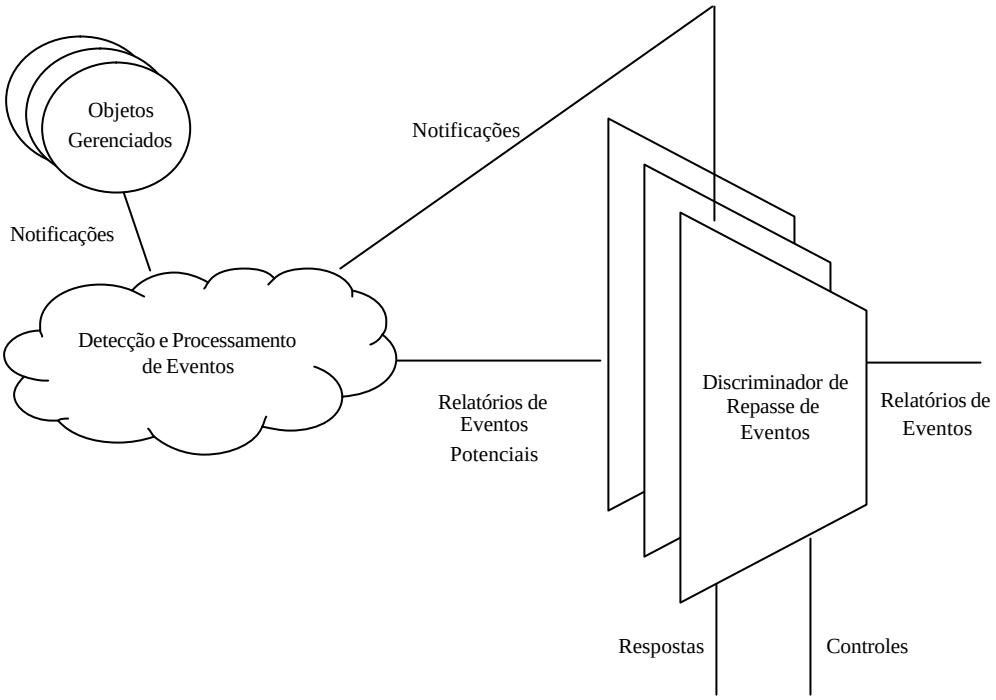


Figura 2.8 - Modelo de gerenciamento de relatórios de eventos

Em linhas gerais, esta função tem os seguintes objetivos:

- Definir um serviço de controle de relatórios de eventos que permita selecionar os relatórios de eventos que devem ser enviados a um sistema de gerenciamento em particular;
- Determinar os destinatários para os quais os relatórios de evento devem ser enviados;
- Especificar um mecanismo para controlar o repasse de relatórios de eventos que permita, por exemplo, suspender ou retomar a transmissão desses relatórios;

- Possibilitar que um sistema de gerenciamento externo modifique as condições usadas na emissão de relatórios de eventos;
- Designar endereços alternativos para onde os relatórios de eventos possam ser encaminhados em caso de indisponibilidade do endereço primário.

A Figura 2.8 ilustra o modelo de gerenciamento de relatórios de eventos e os componentes envolvidos na geração, processamento e emissão desses relatórios.

2.3.3 Função de Controle de *Log*

É usual que em muitas situações os eventos e notificações emitidos pelos objetos gerenciados sejam armazenados, com o objetivo de permitir a análise detalhada de um problema. As funções de gerenciamento também necessitam preservar informações sobre as operações que tenham sido efetuadas sobre um ou mais objetos gerenciados. O repositório onde são armazenadas estas informações é chamado de *Log*.

A Função de Controle de *Log* (LCF - *Log Control Function*) especifica um modelo para controle e manutenção de *Logs*, e tem como finalidade atender os seguintes objetivos:

- Oferecer um serviço de controle de *Logs* flexível, permitindo a seleção de registros preservados por um determinado sistema de gerenciamento;
- Permitir que um sistema de gerenciamento externo modifique os critérios usados para preservação dos registros;
- Permitir que um sistema de gerenciamento externo determine se alguma característica de preservação dos registros foi modificada, ou se algum registro de *Log* foi perdido;
- Oferecer mecanismos que controlem o tempo ou a periodicidade com que a atividade de preservação das informações deve ocorrer, por exemplo, mecanismos para suspender e retomar a atividade de armazenamento dos registros (*Logging*);

- Permitir que um sistema de gerenciamento externo seja capaz de recuperar e eliminar registros de *Logs*, e também criar e eliminar *Logs*.

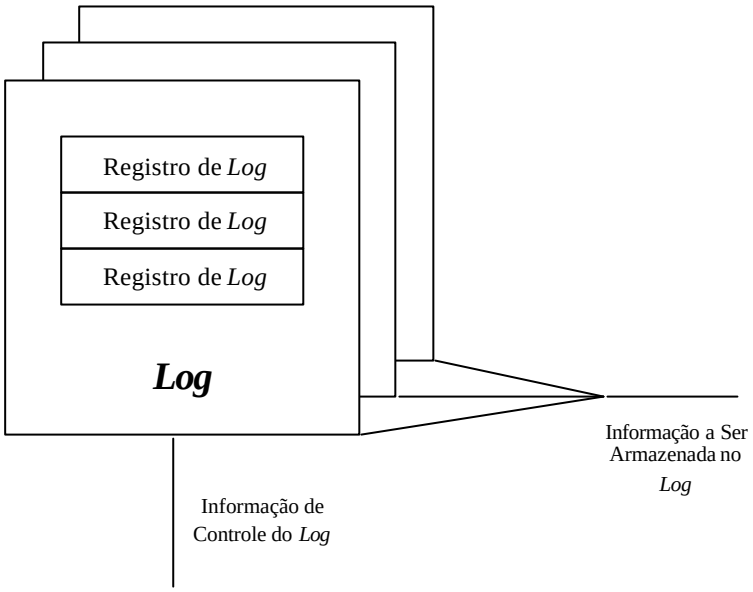


Figura 2.9 - Modelo esquemático do *Log*

A Figura 2.9 ilustra o modelo esquemático do *Log* e dos registros de *Logs*.

2.3.4 Função de Gerenciamento de Teste

Um teste é utilizado para verificar o estado de funcionamento ou extrair informações estatísticas sobre o desempenho de um recurso gerenciado.

A Função de Gerenciamento de Teste (TMF - *Test Management Function*) especifica um modelo e os mecanismos usados para a realização de testes nos recursos gerenciados do sistema. Utilizando os serviços oferecidos por esta função, os seguintes tipos de testes podem ser realizados:

- *Teste de Conexão*: Permite a realização de testes em uma conexão real ou virtual entre dois recursos;

- *Teste de Conectividade*: Usado para verificar se uma conexão pode ser estabelecida entre dois recursos;
- *Teste de Integridade de Dados*: Usado para verificar se dois recursos podem trocar dados sem que eles sejam corrompidos. É também utilizado para medir o tempo necessário para a transferência correta dos dados;
- *Teste de Fim da Conexão*: Usado para testar a operacionalidade de uma conexão entre um recurso gerenciado e um segundo recurso;
- *Teste de Loopback*: Utilizado para verificar a integridade e a taxa de transmissão de dados transmitidos e recebidos através de um caminho de comunicação;

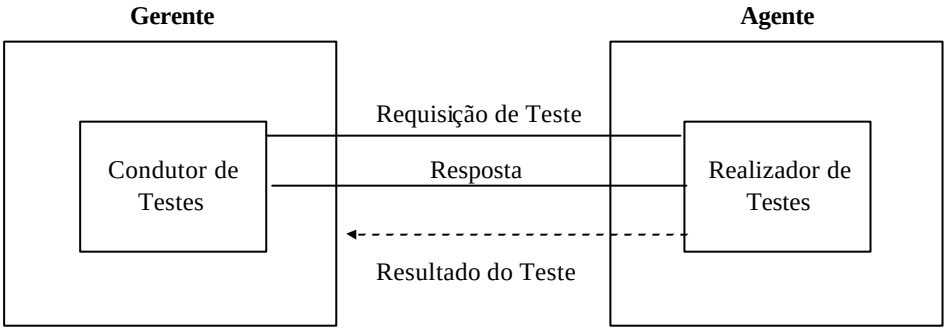


Figura 2.10 - Modelo genérico de testes

- *Teste de Integridade do Protocolo*: Usado para testar se um recurso gerenciado pode estabelecer uma interação através de um certo protocolo;
- *Teste de Limites do Recurso*: Usado para verificar o corretismo do funcionamento de um recurso. O teste é realizado observando a interação entre o recurso e o ambiente em que ele está inserido;
- *Auto-Teste de um Recurso*: Usando para que o próprio recurso realize uma verificação da execução de um de seus procedimentos;

- *Teste da Infra-estrutura de Testes*: Usado para verificar a capacidade de um sistema realizar testes, produzir relatórios de resultados, e responder a monitoramento ou ações de controle.

O modelo genérico para realização de testes empregado por esta função é mostrado pela Figura 2.10. O gerente mantém um processo de aplicação chamado de *Condutor de Testes*, que é responsável por emitir requisições de um dos testes descritos anteriormente. As requisições de testes são recebidas e respondidas por um outro processo de aplicação, que reside em um agente, chamado de *Realizador de Testes*.

2.4 Deficiências do Gerenciamento de Falhas OSI

O gerenciamento de falhas é o conjunto de atividades empregadas para garantir a disponibilidade da rede, mesmo na presença de falhas em seus componentes ou degradação de seu desempenho. Uma das maiores dificuldades desta tarefa deve-se exatamente ao fato de que suas atividades são mais necessárias no próprio momento em que parte da rede está com problemas ou sem funcionamento. Neste contexto, o gerenciamento de falhas deve ser tolerante a elas, ou seja, a presença de problemas não deve prejudicar a própria tarefa com a qual se espera resolvê-los.

As atividades de gerenciamento se dividem, de um modo geral, em monitoramento e controle dos recursos gerenciados. O monitoramento compreende o conjunto de processos utilizados para obter informações sobre os componentes da rede, tornando-as disponíveis para que os processos de controle modifiquem ou coordenem seu comportamento.

Sistema convencionais de gerenciamento de rede e, particularmente, gerenciamento de falhas, utilizam o modelo gerente/agente como suporte às atividades de monitoramento e controle da rede, como discutido nas seções 2.2 e 2.3.

Numa possível implementação prática deste modelo, uma estação fixa exerce a função do gerente e coordena um conjunto de estações agentes, enviando operações de controle e recebendo de volta informações sobre os recursos gerenciados. Esta forma de

implementação, entretanto, é estritamente não confiável, visto que se houver uma falha na estação gerente, todas as atividades de gerenciamento da rede inteira são paralisadas.

Implementações que utilizam um modelo hierárquico entre gerentes e agentes são bastante usuais, nas quais as estações que compõem o sistema de gerenciamento são organizadas na forma de uma árvore. Nestas implementações, a estação *raiz* é o gerente principal, e as estações que compõem as *folhas* exercem os papéis de agentes. As estações intermediárias entre a raiz e as folhas exercem ambos os papéis de gerente e agente. Esta forma de implementação hierárquica também não é confiável, visto que a falha de uma estação intermediária pode paralisar as ações de gerenciamento em uma porção considerável da rede.

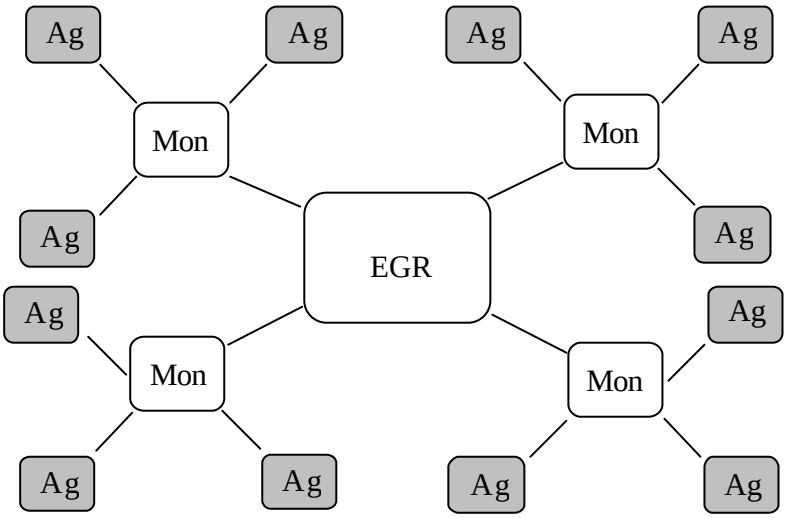


Figura 2.11 - Abordagem usual para implementação do sistema de gerenciamento

A Figura 2.11 ilustra uma abordagem usual para implementação de um sistema de gerenciamento. Nesta mesma figura podemos observar uma Estação de Gerência de Redes (*EGR*), que funciona como interface e unidade gerente principal do sistema de gerenciamento. As estações *Ag* executam os processos agentes dentro do sistema, atuando diretamente nos recursos gerenciados. As estações *Mon* podem exercer tantos os

papéis de gerentes quanto de agentes, servindo neste último caso aos gerentes de mais alto nível.

Como discutido, este modelo apresenta algumas deficiências para o gerenciamento de falhas, visto que algumas configurações de falhas podem causar a paralisação das ações de gerenciamento em toda ou grande parte da rede. Além desses fatores, cada um dos gerentes deve ser responsável pela realização de testes em um grande número de componentes, o que agrega grande complexidade à tarefa de diagnóstico.

A área de *Diagnóstico Automático de Sistema* tem alcançado grandes avanços nos últimos anos, e vários trabalhos têm apresentado algoritmos práticos para detecção e localização de falhas em redes de computadores. Um dos principais motivadores para o uso destes algoritmos é que eles são tolerantes à existência de falhas em qualquer número de componentes da rede, permitindo que os componentes normais remanescentes sejam sempre capazes de obter um diagnóstico correto, o que não ocorre na estratégia convencional de gerenciamento de falhas utilizando o modelo gerente/agente.

CAPÍTULO 3

Domínio do Problema e Pesquisas Relacionadas

3.1 Introdução

Neste capítulo, fazemos uma revisão das pesquisas desenvolvidas na área de algoritmos para *diagnóstico automático de sistema.* Nele são apresentadas as principais contribuições destas pesquisas que são relevantes ao nosso trabalho.

3.2 Diagnóstico Automático de Sistema[1]

Considere um sistema composto de *N* unidades indivisíveis e autônomas, não necessariamente idênticas, que podem estar em um dos dois *estados de funcionamento*: *normal* ou *falha*. O objetivo do diagnóstico automático de sistema é identificar de forma precisa os estados de funcionamento destas unidades.

Durante muitos anos, vários pesquisadores têm empreendido esforços no sentido de propor modelos e estratégias para detecção e identificação de unidades falhas nestes sistemas, cujas estruturas podem ser representadas através de grafos. Nestes grafos, os vértices representam suas unidades autônomas e as arestas representam uma topologia qualquer de interligação entre elas.

[1] *System-level Diagnosis*, na literatura técnica de língua inglesa.

Um modelo de falhas para um sistema qualquer é um conjunto de suposições, ou restrições, que, em linhas gerais, definem como as unidades falhas e as normais se comportam, como os testes sobre as unidades são realizados e seus resultados são obtidos, e ainda como o diagnóstico das unidades é decidido.

O primeiro modelo para diagnosticar falhas nesta classe de sistemas foi proposto no trabalho de Preparata, Metze e Chien. [PRE 67]

Neste, que é considerado o trabalho seminal da teoria de diagnóstico automático de sistema, foram estabelecidas suas bases teóricas, e sua formalização matemática, possibilitando a compreensão dos seus limites, na solução do problema prático da identificação automática de falhas múltiplas, em sistemas cujas unidades são capazes de realizar testes umas sobre as outras.

No chamado modelo PMC[1], ou *modelo de invalidação simétrica*[2], o sistema, denotado por *S*, é representado através de um grafo orientado, no qual uma aresta com peso binário partindo do vértice *i* para o vértice *j* representa um teste realizado pela unidade *i* sobre a unidade *j*, e o peso binário representa o resultado deste teste. De fato, se a unidade *i* testa a unidade *j*, há uma ligação física que conecta *i* e *j*.

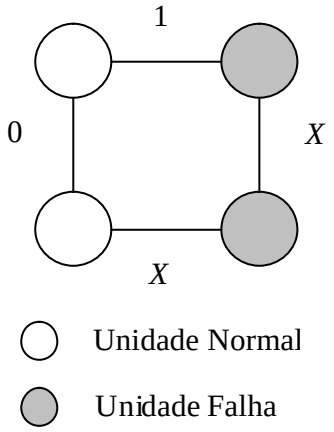


Figura 3.1 - Modelo de testes PMC

[1] Das iniciais dos autores.

[2] Existem pelo menos mais dois outros modelos na literatura técnica, os quais não descreveremos aqui: o modelo de invalidação assimétrica [BAR 76] e o modelo baseado em comparação [MAL 80].

O resultado de um teste realizado pela unidade *i* sobre a unidade *j* é um valor binário: 1 se o teste falhou, e 0, caso contrário, e depende tanto do estado de *i* quanto de *j*, como mostra a Figura 3.1. Nesta figura, vemos que os resultados dos testes realizados por uma unidade normal são corretos, isto é, indicam precisamente os estados de funcionamento das unidades testadas. Os resultados dos testes realizados por uma unidade falha são arbitrários, e estão representados na figura por *X*, significando que podem tanto ter valor 1 quanto 0, independente dos estados das unidades que estão sendo testadas. Isto caracteriza que as falhas do modelo PMC sejam todas do tipo *Bizantina*. O conjunto de todos os resultados dos testes é denominado *síndrome*[1].

A *situação de falha* de um sistema é o conjunto dos estados de todas as suas unidades. O procedimento de diagnóstico neste modelo é a determinação da situação de falha de um sistema a partir da sua síndrome. No modelo PMC, pressupõe-se a existência de um *observador central*, responsável por receber todos os resultados dos testes (síndrome), e determinar o diagnóstico dos estados de funcionamento de todas as unidades do sistema.

A Figura 3.2 ilustra uma topologia de testes para um sistema *S*, composto por seis unidades, onde vemos sua situação de falha e uma possível síndrome.

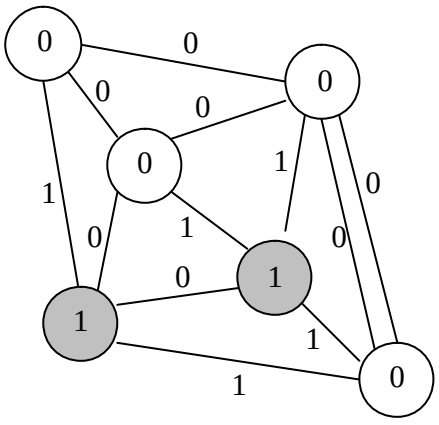


Figura 3.2 - Exemplo de sistema no modelo PMC

[1] *Syndrome*, na literatura técnica de língua inglesa.

Para que o modelo seja confiável, assume-se que todas as falhas são permanentes, ou seja, falhas *transitórias* ou *intermitentes* não são tratadas. Falhas transitórias são aquelas que ocorrem por um período de tempo curto, fazendo com que, possivelmente, uma unidade seja considerada falha, quando na verdade está normal. Por outro lado, uma falha intermitente é aquela que ocorre quando uma unidade está falha, mas pode, em certos intervalos, se comportar como uma unidade normal, e assim ser incorretamente considerada como tal, após um teste realizado sobre ela. O modelo não considera estes tipos de falha, mas, na prática, falhas transitórias e intermitentes são mais comuns, e mais difíceis de diagnosticar, do que as falhas permanentes. [SIE 92]

Preparata *et al.* [PRE 67] definiram o conceito de *t-diagnosticabilidade*[1] como sendo a possibilidade de diagnosticar uma situação de falha com t ou menos unidades falhas, dada uma síndrome do sistema. Eles provaram que o sistema deve conter n unidades, onde $n \geq 2t + 1$, e, além disso, cada unidade deve ser testada por pelo menos t outras unidades distintas. Foi mostrado ainda que, se $n \geq 2t + 1$, é sempre possível compor um sistema de conexões S, tal que S seja *t-diagnosticável*[2].

Dois casos em especial são estudados em detalhes em [PRE 67], conforme o diagnóstico seja realizado juntamente com reparação de unidades falhas ou não. Em um sistema onde não é factível obter-se o diagnóstico em múltiplas fases, todas as unidades falhas devem ser diagnosticadas corretamente em um único passo de testes. Isto foi definido como *diagnóstico de um passo*, ou *diagnóstico sem reparação.* Por outro lado, se o sistema é reparável, então apenas uma unidade falha necessita ser identificada, se existir. Uma vez identificada, esta unidade falha pode ser reparada, e o processo de diagnóstico continua até que outra unidade falha seja identificada e reparada também. Isto foi definido como *diagnóstico sequencial*, ou *diagnóstico com reparação.*

O trabalho de Preparata *et al.* apresenta as condições necessárias para a identificação de todas as unidades falhas, em um sistema com capacidade de diagnóstico automático.

---

[1] *t-diagnosability*, no trabalho original.

[2] *t-diagnosable*, no trabalho original.

Uma questão natural é saber, além das condições necessárias, quais condições são suficientes para que um sistema *S* seja *t-diagnosticável*, dado um certo valor de *t* (o limite máximo do número de unidades falhas permitidas). Este problema é conhecido como *problema da caracterização* de sistemas diagnosticáveis.

Hakimi e Amin [HAK 74] mostraram que as condições necessárias apresentadas em [PRE 67] são também suficientes, desde que no sistema não haja duas unidades que testem-se mutuamente. Eles apresentaram uma solução para sistemas *t-diagnosticáveis* em um único passo de testes. O número de unidades nestes sistemas é $n \geq 2t + 1$, e é necessário que cada unidade seja testada por pelo menos *t* outras unidades distintas. Além disso, eles mostraram que para cada inteiro p, $0 \leq p \leq t$, cada subconjunto V de unidades, $|V| = n - 2t + p$, deve ser testado por mais do que p unidades que estão fora de V.

Dado um conjunto de unidades e uma topologia de testes qualquer entre elas, o problema de saber qual o número máximo *t* destas unidades que podem falhar de modo que ainda seja possível identificá-las *unicamente* a partir de qualquer síndrome obtida é conhecido como problema do cálculo da *diagnosticabilidade*[1].

Sullivan [SUL 84] apresentou uma solução para o problema do cálculo da *diagnosticabilidade*, dentro das suposições do modelo PMC. Ele desenvolveu um algoritmo com complexidade $\boldsymbol{O}[|E|n^{1.5}]$, onde |E| é o número de testes, que pode calcular a *t-diagnosticabilidade* de uma dada topologia de testes em um sistema com *n* unidades.

3.3 Diagnóstico Adaptativo

Os primeiros trabalhos na teoria de sistemas *t-diagnosticáveis* com *n* unidades assumiam a existência de topologias fixas de testes, usadas como base para a obtenção das síndromes, e posterior identificação de suas unidades falhas, desde que o número destas unidades não excedesse *t*.

[1] Este termo não existe na norma oficial corrente da língua portuguesa.

Nakajima [NAK 81] foi o primeiro a propor uma estratégia diferente, supondo que cada unidade é capaz de realizar testes em todas as outras. Neste caso, a topologia de testes seria modificada adaptativamente, até que uma unidade normal fosse encontrada. Esta unidade normal seria então usada para testar as demais e, consequentemente, todas as unidades falhas seriam identificadas. Neste trabalho, foi mostrado que uma unidade normal seria identificada com os resultados de $t(t + 1)$ testes, e, deste modo, com $(n - 1)$ testes adicionais, realizados por esta unidade normal, seriam identificadas todas as unidades falhas.

Posteriormente, Hakimi e Nakajima [HAK 84] exploraram a idéia apresentada em [NAK 81] e propuseram um algoritmo adaptativo para o modelo PMC, capaz de identificar uma unidade normal após a aplicação de no máximo $(2t - 1)$ testes, dado que o número de unidades falhas não fosse superior a t. Este resultado implica que todas as unidades falhas seriam identificadas com no máximo $(2t - 1) + (n - 1)$ testes.

A idéia geral do trabalho é a seguinte: dado um sistema S com n unidades, onde há uma ligação entre quaisquer duas delas e o número máximo de unidades falhas é limitado a t, com $n \geq 2t + 1$, o algoritmo procura uma unidade u e uma seqüência de t outras unidades distintas $v_1, ..., v_t$, tal que o resultado do teste realizado por v_t sobre u $(v_t \rightarrow u)$ é 0, e os resultados dos testes realizados por v_i sobre v_{i+1} $(v_i \rightarrow v_{i+1})$ são todos 0, onde $1 \leq i \leq (t - 1)$. Neste caso, garante-se que a unidade u está normal.

A Figura 3.3 ilustra um sistema com cinco unidades, completamente conectadas, capazes de testar umas as outras, como ilustrado pelas arestas apontando em ambas as direções. Suponha que o número máximo (t) de unidades falhas neste sistema é 2, e as unidades *1* e *5* estão falhas.

O algoritmo começa escolhendo aleatoriamente uma unidade u do sistema, e, em seguida, procura uma seqüência de outras unidades v_i, satisfazendo as condições descritas anteriormente. Suponha que, por exemplo, a unidade *2* foi escolhida, e que os resultados dos testes realizados por *5* e *3* são 0. Assim, pode-se afirmar com certeza que *2* é normal, pois o algoritmo encontrou uma seqüência: $5 \rightarrow 3 \rightarrow 2$, tal que os resultados destes testes são todos iguais a 0. Este raciocínio está ilustrado na Figura 3.4, considerando todos os

possíveis casos em que os resultados destes testes seriam 0, dados os estados de funcionamento das unidades envolvidas.

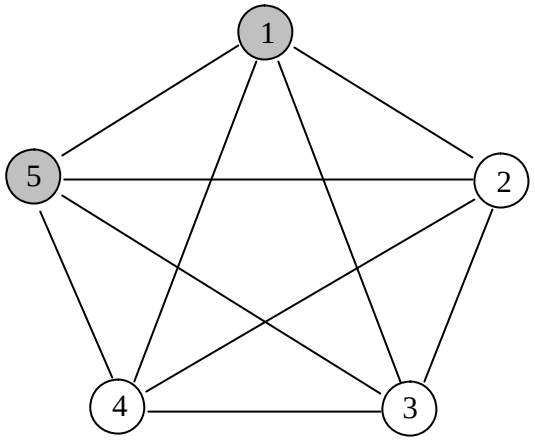


Figura 3.3 - Diagnóstico adaptativo em sistema com cinco unidades

Através da Figura 3.4 podemos observar que apenas nos casos 1,4,7 e 8, os dois testes realizados na seqüência $v_1 \rightarrow v_2 \rightarrow u$ podem ambos ser 0. O Caso 7 não é permitido no sistema, visto que assume-se que o número máximo de falhas (t) é limitado a 2. Deste modo, se os dois testes são 0, u necessariamente deve ser normal.

Caso	v_1	$\rightarrow$	v_2	$\rightarrow$	u
1	F	0/1	F	0/1	N
2	F	0/1	N	1	F
3	N	1	F	0/1	F
4	F	0/1	N	0	N
5	N	1	F	0/1	N
6	N	0	N	1	F
7	F	0/1	F	0/1	F
8	N	0	N	0	N

Figura 3.4 - Casos possíveis para três unidades

O algoritmo é adaptativo no sentido de que, enquanto esta seqüência de testes iguais a zero não for encontrada, novos testes são escolhidos, e a execução continua.

Em [HAK 84] mostra-se que o algoritmo encontra uma unidade normal em no máximo ($2t$ - 1) testes adaptativos. Neste caso, encontrada esta unidade, ela pode ser usada para encontrar todas as unidades falhas em no máximo (n - 1) testes adicionais, o que leva a um total de (n + $2t$ - 2) testes no pior caso, um resultado melhor do que o apresentado originalmente em [NAK 81].

No trabalho, é considerada ainda uma aplicação do algoritmo para localização de uma unidade falha em sistemas de diagnóstico sequencial, ou diagnóstico com reparação. Neste caso, mostra-se que o algoritmo é capaz de encontrar uma unidade *falha* em no máximo n testes adaptativos.

3.4 Diagnóstico Distribuído

O conceito de diagnóstico distribuído foi introduzido por Kuhl e Reddy. [KUH 80,81] Antes desta proposta, pressupunha-se a existência de um *observador central*, responsável por coletar todos os resultados dos testes realizados nas unidades do sistema, e também executar um procedimento para diagnosticar os seus estados de funcionamento.

Este tipo de abordagem implica altos custos em termos de tempo, troca de informações, e, principalmente, confiabilidade, pois este modelo depende intrinsecamente da disponibilidade do *observador central,* que, em caso de falha, impede a obtenção do diagnóstico do sistema. Além do mais, em sistemas distribuídos, é mais natural que esta atividade seja compartilhada através de todos os elementos computacionais, de forma onde não haja um único ponto que, em caso de falha, impeça a porção restante do sistema de conhecer o seu estado.

Kuhl e Reddy apresentaram o algoritmo SELF para diagnóstico distribuído, usando uma estratégia na qual as unidades normais testam e recebem, através de unidades vizinhas, relatórios com resultados de testes realizados em outras unidades que não testam diretamente. Desta forma, cada uma delas pode realizar um diagnóstico

consistente, de forma independente. No algoritmo, assume-se que o número de unidades falhas é limitado por um máximo predefinido *t*. A topologia de testes é fixa, ou seja, não adaptativa, de modo que cada unidade testa um conjunto constante de suas vizinhas. Unidades normais propagam relatórios de testes através de suas vizinhas, de modo que estes resultados alcancem todas as outras unidades do sistema. Nenhuma restrição é imposta ao comportamento de unidades falhas, que podem propagar resultados arbitrários de testes que realizem.

Posteriormente, Hosseini *et al.* [HOS 84] apresentaram o algoritmo NEW_SELF, que também trabalha com uma topologia de testes fixa, no entanto, possui uma característica especial na qual uma unidade pode falhar e ser automaticamente reintegrada ao sistema de diagnóstico, após ser reparada, enquanto o algoritmo está sendo executado. Algoritmos com esta característica são conhecidos na literatura técnica como algoritmos *on-line*[1]. À exemplo do algoritmo SELF, relatórios com resultados de testes são propagados através de unidades vizinhas, de modo que alcancem todas as outras unidades do sistema.

O algoritmo NEW_SELF garante a confiabilidade dos relatórios de testes, restringindo que a propagação destes relatórios ocorra apenas entre as unidades normais. Uma unidade aceita informações provenientes de uma unidade vizinha apenas se ela determinar que esta vizinha está normal. A Figura 3.5 ilustra um exemplo da estratégia utilizada pelo algoritmo NEW_SELF, em que uma unidade recebe resultados de testes realizados por outras unidades através de suas vizinhas. Nesta figura, vemos que as unidades *4* e *5* estão falhas e as demais estão normais. A unidade *1* testa a unidade *2* e determina que ela está normal. Desta forma, *1* assume que os relatórios de testes provenientes de *2* são confiáveis, e pode diagnosticar com precisão que a unidade *3* está normal enquanto a unidade *4* está falha. Por outro lado, *1* não aceita nenhum relatório da unidade *5*, pois esta unidade está falha. Deste modo, *1* não pode diagnosticar o estado da unidade *6*, usando relatórios provenientes da unidade *5*.

[1] Usaremos esta expressão original em inglês ao longo do texto.

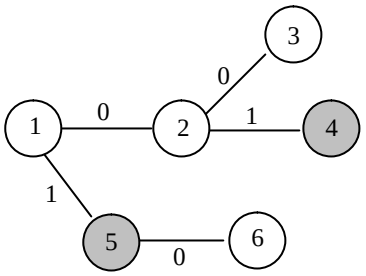


Figura 3.5 - Propagação de relatórios através de unidades normais

O seguinte mecanismo é utilizado para a troca de relatórios de resultados de testes entre duas unidades vizinhas u_i e u_j, nos algoritmos NEW_SELF:

1) u_i testa u_j e verifica que ela está normal;

2) u_i recebe os relatórios dos resultados de testes de u_j;

3) u_i testa u_j e verifica que ela está normal;

4) u_i assume que as informações recebidas no passo 2 são válidas.

Este mecanismo assume que a unidade u_j não pode falhar e ser reparada sem que isto seja detectado por u_i, durante o intervalo entre os dois testes consecutivos realizados nos passos 1) e 3).

Para que o algoritmo obtenha um diagnóstico correto é necessário que cada unidade normal receba os relatórios dos resultados dos testes realizados por todas as outras unidades normais. Em [HOS 84] mostra-se que esta restrição impõe que cada unidade deva ser testada por pelo menos $(t + 1)$ outras unidades, onde t é o limite no número de unidades falhas permissíveis. Nestas condições, verifica-se que cada unidade propaga resultados dos testes que realiza para pelo menos uma outra unidade normal.

Para um sistema *t-diagnosticável* com n unidades, o algoritmo NEW_SELF requer pelo menos $n(t + 1)$ testes, desde que cada unidade é testada por pelo menos $(t + 1)$ outras. O número de mensagens necessário para propagar todos os relatórios com os resultados dos testes realizados é igual a $n^2(t + 1)^2$.

Bianchini *et al.* [BIA 90] verificaram que a aplicação do algoritmo NEW_SELF em redes reais implicaria em custos de tráfego inaceitáveis, decorrentes do número substancial de mensagens necessárias à sua execução. Em seu trabalho, é proposta uma adaptação do algoritmo NEW_SELF, que os autores chamaram de EVENT_SELF, onde uma técnica de propagação de relatórios *movida a eventos* [1] é empregada.

A técnica de propagação de relatórios *movida a eventos* emprega uma estratégia na qual apenas os testes são realizados em intervalos regulares, e nenhum relatório é enviado a menos que um novo evento de falha ou reparação ocorra. No trabalho de Bianchini, prova-se que há apenas duas situações em que um novo relatório deve ser propagado para uma unidade vizinha. A primeira situação ocorre quando o resultado de um teste realizado por uma unidade u sobre a unidade k difere do resultado do teste realizado anteriormente, significando que a unidade k falhou ou foi reparada. Neste caso, um relatório para este evento é enviado para todas as unidades que testam u. A segunda situação ocorre quando uma unidade k havia diagnosticado que uma das unidades que a testam, digamos u, estava falha, e então recebe um relatório informando que u está normal. Nesta situação, k deve novamente propagar todos os seus relatórios para u, a fim de que u também possa realizar seu procedimento de diagnóstico corretamente.

Usando esta abordagem, o número de mensagens necessárias para um diagnóstico correto é significativamente reduzido comparativamente ao algoritmo NEW_SELF. Se f é o número de unidades falhas do sistema, e denotamos por Δf o número de mudanças de estado ocorridas nestas unidades, o algoritmo EVENT_SELF requer um total de Δfnt^2 mensagens para um sistema *t-diagnosticável* com n unidades.

O algoritmo EVENT_SELF também apresenta uma melhoria substancial na latência de diagnóstico. A *latência de diagnóstico* é o tempo decorrido desde a detecção de um novo evento de falha ou reparação até o instante em que todas as unidades normais tenham obtido diagnóstico correto para ele. Nos algoritmos SELF, a latência de diagnóstico é igual ao tempo necessário para propagar os relatórios de resultados de

[1] *Event-driven*, no trabalho original.

testes correspondentes aos eventos de falha ou reparação para todas as unidades do sistema.

No algoritmo NEW_SELF, uma nova informação que chega em uma unidade *u*, vinda da unidade *k*, é considerada válida apenas ao fim do intervalo entre os dois testes consecutivos realizados por *u* em *k*, como visto anteriormente. No algoritmo EVENT_SELF, o procedimento de validação da mensagem é iniciado imediatamente, tão logo uma nova mensagem seja recebida. Desta forma, o tempo de validação da mensagem é sempre menor do que o intervalo de dois testes consecutivos, e a latência de diagnóstico é reduzida.

3.5 Algoritmos de Diagnóstico para Redes Completamente Conectadas

3.5.1 Adaptive DSD

Bianchini e Buskens [BIA 92] apresentaram um algoritmo para diagnóstico em redes completamente conectadas, física ou logicamente[1], denominado *Adaptive Distributed System-Level Diagnosis* ou *Adaptive DSD.* Este algoritmo utiliza uma estratégia de diagnóstico que é ao mesmo tempo *adaptativa* e *distribuída.*

Seguindo a literatura, usaremos alternativamente a palavra nó ao invés de unidade e rede ao invés de sistema.

O algoritmo *Adaptive DSD* difere consideravelmente dos algoritmos SELF, dado que sua topologia de testes é adaptativa, e determinada pela situação de falha da rede. Ele funciona de forma *on-line*, ou seja, falhas ou reparações de nós são tratados automaticamente, em tempo de execução, alterando sua topologia de testes a cada novo evento. Não são consideradas falhas nos *enlaces* de comunicação entre os nós. Uma outra diferença substancial em relação aos algoritmos SELF é que o número máximo de nós falhos permitido não é limitado, isto é, os nós normais são capazes de diagnosticar corretamente qualquer número de nós falhos existentes.

[1] Os autores implementaram uma versão do algoritmo para redes *Ethernet* usando os protocolos UDP/IP.

Cada nó normal executa o algoritmo a *intervalos de testes* predefinidos, e dispõe de uma lista circular ordenada com todos os nós da rede. Ao fim de cada um destes intervalos, um nó executa testes seqüencialmente, a partir do primeiro nó consecutivo à sua própria entrada na lista ordenada, até que encontre um outro nó normal, ou verifique que todos estão falhos. Se um outro nó normal é encontrado, as informações de diagnóstico locais são atualizadas com as informações provenientes deste nó.

Uma *etapa de testes*[1] é definida como o período de tempo em que todos os nós normais tenham executado o algoritmo *Adaptive DSD* pelo menos uma vez. Ao fim da primeira etapa de testes, o algoritmo constrói um ciclo orientado de testes entre todos os nós normais. A Figura 3.6 ilustra um exemplo de um sistema com oito nós, dos quais três estão falhos, e o ciclo orientado de testes gerado após a execução do *Adaptive DSD*.

Neste exemplo, os nós *1*, *4* e *5* estão falhos, e os demais estão normais. O algoritmo especifica que um nó realize testes seqüencialmente até que encontre um outro nó normal. Por exemplo, o nó *0* testa o nó *1* verificando que ele está falho e deste modo continua testando. Subseqüentemente, *0* testa *2*, verificando que ele está normal e, deste modo, ele pára com sua seqüência de testes, atualizando suas informações de diagnóstico locais com as informações provenientes do nó *2*. O nó *2*, por sua vez, verifica que *3* está normal e pára com sua seqüência de testes imediatamente. O nó *3* deve testar três outros nós, antes de encontrar um que esteja normal, no caso, *6*.

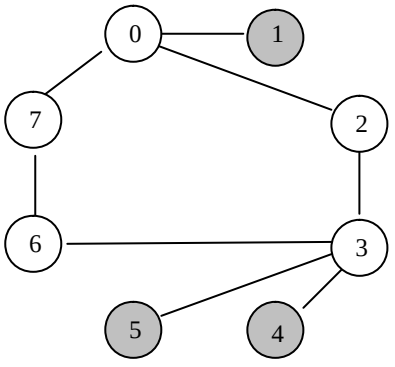


Figura 3.6 - Ciclo orientado de testes entre os nós normais

[1] *Testing round*, no trabalho original.

Cada nó *i* que executa o algoritmo tem um vetor chamado $TESTED\_UP_i$, que contem *N* entradas, uma para cada um dos *N* nós da rede, indexadas por seus identificadores. A entrada $TESTED\_UP_i[k] = j$ significa que o nó *i* recebeu a informação de diagnóstico de um outro nó normal especificando que o nó *k* testou o nó *j* e verificou que *j* estava normal. Uma entrada $TESTED\_UP_i[j]$ é *arbitrária* (*X*) se o nó *j* está falho. A Figura 3.7 mostra o vetor $TESTED\_UP_2$ mantida no nó *2*, do sistema exemplo mostrado pela Figura 3.6.

$TESTED\_UP_2[0] =$	2
$TESTED\_UP_2[1] =$	*X*
$TESTED\_UP_2[2] =$	3
$TESTED\_UP_2[3] =$	6
$TESTED\_UP_2[4] =$	*X*
$TESTED\_UP_2[5] =$	*X*
$TESTED\_UP_2[6] =$	7
$TESTED\_UP_2[7] =$	0

Figura 3.7 - Estrutura de dados mantida no nó *2*

Quando um nó *i* testa um outro nó *j* e verifica que ele está normal, *i* atribui *j* à entrada $TESTED\_UP_i[i]$, e atualiza todas as demais entradas com os valores provenientes de $TESTED\_UP_j$. Usando esta estratégia, se *k* é o predecessor normal imediato de *i*, na próxima etapa de testes, *k* receberá as informações mantidas em *i* e este mecanismo se repete em etapas de testes posteriores, até que todos os nós adquiram esta informação. Desta forma, o vetor TESTED_UP é enviado aos nós na direção inversa dos testes realizados por eles.

O diagnóstico dos estados de funcionamento dos nós da rede é feito usando o vetor TESTED_UP em cada nó independentemente. O procedimento de diagnóstico é realizado seguindo os caminhos entre os nós normais mantidos no vetor TESTED_UP. Se o nó está no caminho, ele está normal, caso contrário está falho. Cada nó *i* constrói um vetor chamado $STATE_i$, onde $STATE_i[j]$ é o estado de funcionamento correto do nó *j*.

Inicialmente, o nó *i* atribui o valor FALHO ao estado de funcionamento de todas as entradas de $STATE_i$. O procedimento de diagnóstico percorre o vetor $TESTED\_UP_i$, a partir da sua própria entrada *i*, atribuindo o valor NORMAL às entradas de $STATE_i$, correspondentes ao caminho de nós normais existente em $TESTED\_UP_i$. Para o vetor mostrado pela Figura 3.7, o nó *2* percorreria o seguinte caminho de nós normais: *3,6,7,0,2*. A Figura 3.8 mostra o vetor $STATE_2$ resultante após o procedimento de diagnóstico ser executado no nó *2*.

$STATE_2[0]$ =	NORMAL
$STATE_2[1]$ =	FALHO
$STATE_2[2]$ =	NORMAL
$STATE_2[3]$ =	NORMAL
$STATE_2[4]$ =	FALHO
$STATE_2[5]$ =	FALHO
$STATE_2[6]$ =	NORMAL
$STATE_2[7]$ =	NORMAL

Figura 3.8 - Vetor de diagnóstico mantido no nó *2*

Em [BIA 92], mostra-se que a latência de diagnóstico do algoritmo *Adaptive DSD* é igual a *N* etapas de testes. Após no máximo *N* etapas de testes, o vetor TESTED_UP é o mesmo em todos os nós normais da rede, dado um evento de falha ou reparação, ocasionando que todos obtenham um diagnóstico correto para o evento. Para redes baseadas em difusão, ou *broadcast* de mensagens, os autores sugerem que técnicas *movidas a eventos* e *multicasting* podem ser usadas objetivando reduzir a latência de diagnóstico.

Em [DUA 96] é apresentada uma implementação do algoritmo *Adaptive DSD*, usando o protocolo padronizado de gerência de redes *Simple Network Management Protocol* (SNMP).

3.5.2 Hierarquical Adaptive DSD

E. P. Duarte e T. Nanya [DUA 98] apresentaram uma evolução do algoritmo *Adaptive DSD*, que reduz a sua latência de diagnóstico original (que é de no máximo N etapas de testes, em uma rede com N nós) para $(log_2 N)^2$ etapas de testes, no pior caso. O algoritmo proposto, chamado de *Hierarquical Adaptive Distributed System-Level Diagnosis* ou *Hi-ADSD*, utiliza uma estratégia na qual os nós são agrupados em *clusters*[1] lógicos de tamanhos progressivos, onde os testes são realizados de forma hierárquica.

A principal motivação do trabalho deve-se ao fato de não haver provas concretas de que a técnica de propagação de relatórios *movida a eventos*, proposta originalmente em [BIA 92] com o objetivo de reduzir a latência (ex. uso de *multicasting* ou *broadcast* após uma falha ser detectada), são eficazes. No trabalho, assume-se, à exemplo do *Adaptive DSD*, uma rede completamente conectada onde não são permitidas falhas nos enlaces de comunicação entre os nós, e o modelo de falhas PMC.

O algoritmo utiliza uma estratégia de testes baseada na técnica de *dividir para conquistar*. [DUA 95] Nesta estratégia, os nós da rede são agrupados em *clusters* lógicos e os testes são realizados nos nós de cada *cluster* de forma hierárquica. O número de nós de um *cluster*, o seu tamanho, é sempre uma potência de dois. Inicialmente, assume-se que N também é uma potência de dois e o próprio sistema é um *cluster* de tamanho N. Um *cluster* de n nós $n_j, ..., n_{j+n-1}$, onde o resto da divisão de j por n é igual a zero, e n é uma potência de dois, é definido recursivamente como sendo um único nó, no caso de $n = 1$, ou a união de dois *clusters*, um contendo os nós $n_j, ..., n_{j+n/2-1}$, e outro contendo os nós $n_{j+n/2}, ..., n_{j+n-1}$. A Figura 3.9 mostra um sistema com oito nós organizados em *clusters*.

Testes são realizados em intervalos regulares. No primeiro intervalo, cada nó realiza testes nos nós de um *cluster* que contem um único nó, no segundo intervalo, nos nós de um *cluster* que contem dois nós, no terceiro, nos nós de um *cluster* que contem

[1] Usaremos esta expressão original em inglês ao longo do texto.

quatro nós, e assim por diante, até o *cluster* de 2^{logN-1}, ou *N/2* nós, ser testado. Após isso, o *cluster* de tamanho 1 é testado novamente e o processo se repete indefinidamente.

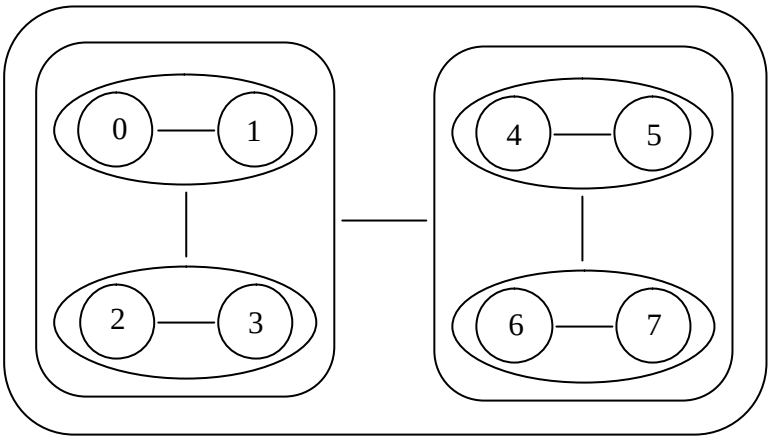


Figura 3.9 - Sistema com oito nós organizados em *clusters*

A lista ordenada de nós testados pelo nó *i* em um *cluster* de tamanho 2^{s-1}, em um dado intervalo de testes, é denotada por $c_{i,s}$. Quando um nó *i* realiza testes nos nós de $c_{i,s}$, ele realiza testes seqüencialmente até encontrar um outro nó normal ou verificar que todos estão falhos. Caso um outro nó normal seja encontrado em $c_{i,s}$, o nó que realizou o teste copia as informações mantidas por este nó normal sobre todos os outros nós de $c_{i,s}$. A Figura 3.10 mostra $c_{i,s}$, $\forall i,s$ para o sistema mostrado na Figura 3.9.

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

Figura 3.10 - $c_{i,s}$ para o sistema da Figura 3.9

Se todos os nós de $c_{i,s}$ estão falhos, o nó *i* prossegue testando os nós de $c_{i,s+1}$ no mesmo intervalo de testes. Novamente, se todos os nós de $c_{i,s+1}$ estão falhos, o nó *i* prossegue testando $c_{i,s+2}$, e assim por diante, até que ele encontre um outro nó normal, ou verifique que todo estão falhos. A Figura 3.11 mostra a hierarquia de testes para oito nós, do ponto de vista do nó *0*. Quando o nó *0* testa um *cluster* de tamanho 2^2, ele primeiro testa o nó *4*. Se o nó *4* está normal, o nó *0* copia a informação de diagnóstico a respeito dos nós *4,5,6* e *7*. Se *4* está falho, o nó *0* testa o nó *5*, e assim por diante.

O algoritmo *Hi-ADSD* utiliza uma árvore para armazenar informações a respeito dos testes em todos os *clusters*. Para diagnosticar efetivamente os estados de todos os nós, é suficiente listar todos os nós da árvore. A Figura 3.12 mostra a árvore mantida no nó *0* para o caso de todos os nós estarem normais.

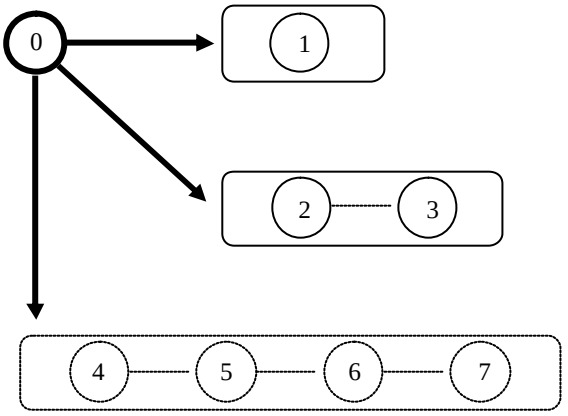


Figura 3.11 - Cada nó testa adaptativamente todos os *clusters*

Assume-se, neste algoritmo, que um nó não pode falhar e ser reparado no intervalo entre dois testes consecutivos realizados sobre ele. No algoritmo *Hi-ADSD* este tempo pode ser de até log_2N etapas de testes, no pior caso. Quando um nó se torna falho e depois é reparado, ele não tem informações de diagnóstico consistentes. O testador deste nó, neste caso, não deve utilizar informações provenientes dele, já que elas podem estar incorretas. Um tempo suficiente, no máximo $(log_2N)^2$ etapas de testes, é necessário para garantir que informações corretas podem ser obtidas novamente, a partir deste nó recém reparado.

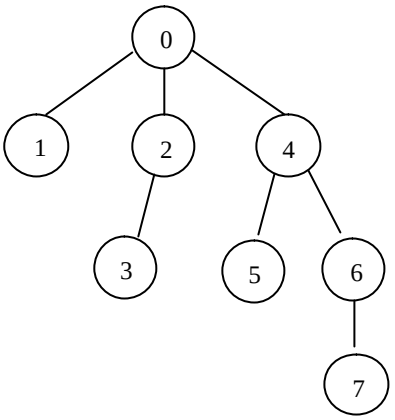


Figura 3.12 - A árvore mantém informações sobre todos os testes

Não é necessário que o número de nós do sistema seja necessariamente uma potência de dois. Caso não seja, de fato, a latência de diagnóstico do algoritmo *Hi-ADSD* é igual a $\lceil (log_2 N)^2 \rceil$, no pior caso.

Os autores apresentam ainda uma implementação do algoritmo *Hi-ADSD* integrada a um Sistema de Gerência de Redes (SGR), baseado no protocolo padronizado de gerência de redes *Simple Network Management Protocol* (SNMP).

3.6 Algoritmos de Diagnóstico para Redes de Topologia Geral

3.6.1 Algoritmo BH

Bagchi e Hakimi [BAG 91] apresentaram um algoritmo para diagnóstico de falhas de processadores em redes de topologia geral, o qual chamaremos de algoritmo *BH.* Inicialmente, cada nó normal conhece apenas quem são os seus vizinhos, e o algoritmo inicia construindo uma topologia de testes baseada numa árvore. As mensagens de diagnóstico são enviadas juntamente com as mensagens que constróem a árvore. Os autores mostraram que o algoritmo é ótimo e requer no máximo $3nlog(p) + O(n + pt)$ mensagens, onde p é o número de processadores normais que iniciam a árvore, e t é o número de processadores falhos. Apesar de ótimo no número de mensagens, o algoritmo

BH não funciona *on-line*, isto é, um nó não pode falhar ou ser reparado durante a execução do algoritmo.

3.6.2 Algoritmo Adapt

Stahl, Buskens e Bianchini [STA 92] apresentaram um algoritmo para diagnosticar falhas de processadores em redes de topologia geral, ao qual chamaram de *Adapt*. O algoritmo *Adapt* funciona de forma distribuída, isto é, cada nó normal é capaz de realizar diagnóstico independentemente. Uma outra característica importante, é que o algoritmo também é adaptativo, ou seja, a topologia de testes entre os nós é modificada de acordo com a situação de falha da rede.

O algoritmo *Adapt* é *on-line*, e funciona corretamente mesmo na presença de falhas nos enlaces de comunicação entre os nós. A topologia de testes é adaptativa, no sentido de que um nó toma decisões localmente com respeito aos testes que irá realizar, baseado tanto nas condições de falha dos demais nós, quanto, indiretamente, nas condições de falha dos enlaces de comunicação entre eles.

Um diagnóstico correto de qualquer número de nós falhos é conseguido, desde que o grafo de interligação entre os nós normais permaneça conexo. Caso este grafo se torne desconexo, devido a uma situação de falha, um diagnóstico correto é conseguido em cada nó apenas para os demais nós pertencentes ao mesmo componente conexo ao qual ele pertença.

O algoritmo assume o modelo de falhas PMC, e, apesar de funcionar *on-line*, que um nó não pode falhar e ser reparado no intervalo entre dois testes consecutivos realizados sobre ele. Um modelo de validação de mensagens semelhante ao usado no algoritmo NEW_SELF [HOS 84] é utilizado para garantir que mensagens recebidas de nós falhos sejam descartadas.

A principal estrutura de dados utilizada é o vetor *Syndromes*, apresentado pela Figura 3.13(a). Uma entrada neste vetor é mantida em cada nó para todos os demais que compõem o sistema de diagnóstico. Cada entrada consiste de uma *timestamp*, seguida por uma lista de nós e seus resultados de testes associados. A *timestamp* é utilizada para

garantir que uma informação antiga seja sobrescrita por uma informação nova. As entradas em cada lista são compostas por um par: identificador de nó/estado de funcionamento. Na figura, *Syndromes*[3] representa a informação fornecida pelo nó n_3, e indica que n_3 está atualmente testando n_1 como normal e n_4 como falho. A entrada x/x em *Syndromes*[4] indica uma informação arbitrária, desde que n_4 está falho. A Figura 3.13(b) mostra a topologia de testes correspondente.

Informações são distribuídas ao longo da rede trocando pacotes entre nós vizinhos. O algoritmo consiste de três fases principais. Imediatamente após a ocorrência de um evento de falha ou reparação, o algoritmo entra na *fase de "Busca"* [1] para reconstruir um grafo de testes fortemente conexo. Uma vez encerrada a fase de *Busca*, o algoritmo inicia a *fase de "Destruição"* [2], na qual todos os testes redundantes são removidos, ou seja, um grafo orientado de testes fortemente conexo mínimo é obtido nesta fase. Finalmente, a *fase de "Informação"* [3] é realizada para atualizar todos os nós com a nova topologia de testes.

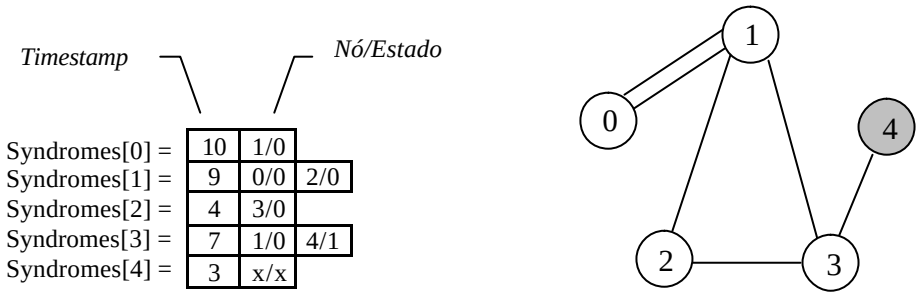


Figura 3.13 - (a) Exemplo de um vetor *Syndromes* **(b) Topologia de testes associada**

O algoritmo constrói uma topologia de testes objetivando garantir que há um caminho partindo de qualquer nó normal para qualquer outro nó (inclusive falho). Testes adaptativos são usados para manter um grafo de testes fortemente conexo mínimo. O

[1] *Search Phase*, no trabalho original.

[2] *Destroy Phase*, no trabalho original.

[3] *Inform Phase*, no trabalho original.

procedimento de *"Busca"* realizado por um nó *n* qualquer é uma versão para dígrafos do algoritmo de caminho mínimo de Dijkstra [BON 76] sobre a topologia de testes inferida a partir do vetor *Syndromes*. Se um nó *n* não pode encontrar um caminho orientado para qualquer um dos seus vizinhos, testes para estes vizinhos são adicionados localmente e a entrada *n.Syndromes*[*n*] é atualizada de acordo. Após o procedimento de "*Busca*" ter sido executado em cada nó, o grafo de testes é garantido ser fortemente conexo. A fase de "*Busca*" é executada em paralelo, isto é, todos os nós realizam este procedimento ao mesmo tempo. A fase de "*Destruição*", na qual os testes desnecessários são removidos, é executada seqüencialmente, para evitar que um nó não remova um teste que um outro nó necessite para conectividade.

As informações são trocadas entre os nós através de pacotes, sempre que um novo evento de falha ou reparação é detectado. Um mecanismo de comparação das informações presentes nestes pacotes é utilizado para garantir que apenas as informações mais recentes sejam processadas e tratadas por cada nó.

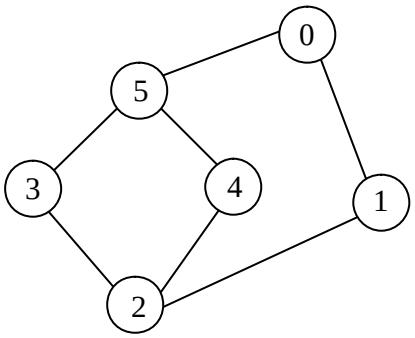
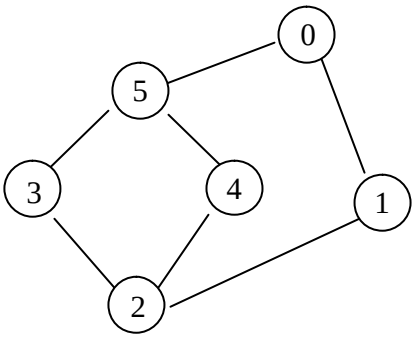


Figura 3.14 - (a) Grafo do sistema



(b) Topologia de testes inicial

Um exemplo da execução do algoritmo *Adapt* é dado pelas Figuras 3.14 e 3.15. O sistema consiste de seis nós, Figura 3.14(a), e um grafo de testes fortemente conexo inicial, Figura 3.14(b). Assumindo que o nó *5* falhe, a fase de "*Busca*" é iniciada, de forma que um novo grafo de testes fortemente conexo seja obtido, Figura 3.15(a). Após a fase de "*Destruição*", o grafo mínimo de testes fortemente conexo é obtido, Figura 3.15(b).

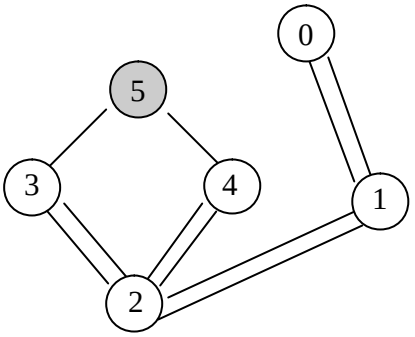
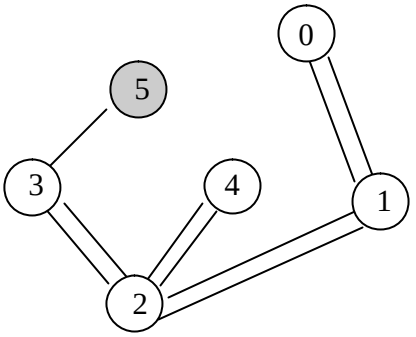


Figura 3.15 - (a) Novo grafo de testes



(b) Grafo de testes reduzido

Os autores analisaram o desempenho do algoritmo *Adapt*, verificando os melhores e piores casos para *tráfego de pacotes*, *número de testes* e *latência de diagnóstico*. Os dados referentes a estes casos para uma rede com N nós e L enlaces estão mostrados na Figura 3.16.

	Tráfego de Pacotes	***Número de Testes***	***Latência de Diagnóstico***
Melhor Caso	$3(2N - 3)$	N	N
Pior Caso	$\boldsymbol{O}(N^2)$	$2L$	$\boldsymbol{O}(N^2)$

Figura 3.16 - Melhores e piores casos para os parâmetros de desempenho do algoritmo *Adapt*

3.6.3 Algoritmo RDZ

Descreveremos este algoritmo com mais detalhes, já que suas principais idéias são usadas no nosso algoritmo.

Rangarajan, Dahbura e Ziegler [RAN 95] apresentaram um algoritmo para diagnóstico distribuído de falhas em processadores de uma rede de topologia geral, ao qual chamaremos de *RDZ*.

No algoritmo *RDZ*, os nós realizam testes periódicos em seus vizinhos, de modo que cada nó normal seja testado por exatamente um único outro nó. O algoritmo emprega a troca de mensagens em paralelo entre os nós para notificar eventos de falha ou reparação. Desta forma, o *overhead* de tráfego devido à execução do algoritmo é mínimo quando os estados dos nós não mudam (apenas testes são realizados), e uma nova informação é propagada tão rápido quanto possível através da rede após ser detectada, devido à propagação de mensagens em paralelo, o que torna o algoritmo *RDZ* ótimo na latência de diagnóstico.

Cada nó normal é o responsável por garantir que exatamente um único dos seus vizinhos o está testando. Desta forma, a topologia de testes no algoritmo *RDZ* também é ótima, empregando o número mínimo de testes possível.

O algoritmo *RDZ* é *on-line*, operando corretamente diante de falhas e reparações apenas de nós enquanto é executado, e apresenta uma vantagem adicional na qual um nó pode falhar e ser reparado durante o intervalo entre dois testes consecutivos realizados sobre ele, sem prejuízo para o funcionamento correto do algoritmo. Em alguns algoritmos anteriores, caso ocorram ambas as transições: normal para falho e em seguida falho para normal, durante o intervalo entre dois testes consecutivos, estas circunstâncias de falha não seriam detectadas, podendo levar a um diagnóstico incorreto. No algoritmo *RDZ* este problema é resolvido através da execução de um procedimento em que um nó recém reparado necessariamente alerta os seus vizinhos sobre sua reparação. Em outras palavras, um nó recém reparado não precisa contar com que algum vizinho primeiramente detecte que ele foi reparado e propague uma informação a respeito disso. Este mecanismo permite ainda que um nó falho não precise continuar sendo testado periodicamente.

No modelo de falhas adotado pelo algoritmo *RDZ* assume-se que um nó normal é capaz de realizar testes em um vizinho e responder corretamente a um teste realizado sobre ele dentro de um tempo máximo preestabelecido, e também solicitar que um vizinho seu passe a testá-lo, se for necessário. Em contraste, assume-se que um nó falho não é capaz de responder a um teste, nem a uma requisição para testar um vizinho. Um nó falho também não propaga mensagens de diagnóstico enviadas para ele, tampouco

gera mensagens de diagnóstico espúrias. Em outras palavras, assume-se que um nó falho simplesmente cessa sua operação, e não é capaz de alertar outros nós sobre sua falha. Este modelo de falhas é conhecido na literatura como *fail-stop model*. [SCH 83]

Falhas nos enlaces de comunicação não são tratadas pelo algoritmo *RDZ*, ou seja, um nó pode ser dado como falho se, de fato, o teste realizado sobre ele falhou devido a uma falha em um de seus enlaces, mesmo que haja um outro caminho alternativo para alcançá-lo na rede. Assim, para que o algoritmo obtenha os resultados desejados, pressupõe-se que todos enlaces de comunicação estejam funcionando corretamente. Isto ocorre por que o algoritmo utiliza uma topologia mínima para testes, e não é possível perceber a diferença entre a falha de um nó e de um enlace.

Os nós detectam falhas em vizinhos e então propagam estas informações para outros nós da rede em duas etapas distintas: *detecção* e *disseminação*. A informação de diagnóstico que um nó propaga consiste de *contadores de eventos de transição de estado*, onde um evento de transição de estado é definido como sendo uma transição realizada por um nó de normal para falho ou de falho para normal.

Quando um evento de falha ou reparação é detectado, inicia-se a etapa de *disseminação.* Nesta etapa, a informação sobre o novo evento é enviada pelo nó que a detectou a todos os seus vizinhos, estes, por sua vez, também propagam a informação para todos os seus vizinhos e assim sucessivamente, de modo que, eventualmente, todos os nós normais atingíveis, dada a situação de falha atual, recebam esta informação.

O mecanismo de *validação de transação*[1] é apresentado. Um nó propaga informações para um vizinho utilizando este mecanismo. Quando um nó *i* propaga uma informação para outro nó *j* é necessário que *i* determine se *j* está normal. Deste modo, quando *j* recebe e processa a informação de *i* ele deve enviar uma confirmação de recebimento para *i.* Esta mensagem de confirmação é uma função do conteúdo da mensagem original. Deste modo, se *j* já está falho ou falhar após receber esta mensagem, *i* vai exceder o tempo máximo de espera por uma mensagem de confirmação (*time-out*), ou receber uma mensagem de confirmação inválida, concluindo corretamente que há um

[1] *Validating transaction*, no trabalho original.

evento de falha em *j*. Em outras palavras, a propagação de mensagens utilizando o conceito de *validação de transação*, também serve, indiretamente, como um teste.

Quando um nó *j* recebe uma informação de um vizinho *i* ele compara o conteúdo desta mensagem com sua informação local, e a classifica em três casos: *mesma, nova* ou *antiga*. Se a informação é a *mesma* significa que *i* e *j* têm as mesmas informações sobre todos os nós da rede, neste caso, esta mensagem é simplesmente ignorada, e não é propagada adiante por *j*. Se a informação é *antiga*, então *j* tem pelo menos uma informação mais recente do que *i* sobre algum outro nó da rede, neste caso, *j* envia uma informação de volta apenas para *i* contendo sua informação local. Se a informação é *nova*, então *j* tem pelo menos uma informação mais antiga do que *i* sobre algum outro nó da rede. Neste caso, *j* atualiza suas informações locais para o que há de mais atual sobre cada nó da rede, e compõe uma nova mensagem com estas informações mais atualizadas propagando-a em seguida para todos os seus vizinhos, inclusive *i*.

As duas estruturas de dados mantidas em cada nó *i* são as seguintes:

1) Um vetor de contadores de eventos de todos os *N* nós da rede, denotado por $event_i[1...N]$, onde $event_i[j]$ contém o valor de um contador representando o mais recente evento de falha ou reparação detectado em *j* através dos testes realizados sobre ele, dadas as mensagens que chegaram em *i* até este momento. Cada entrada *j* deste vetor é sempre incrementada de uma unidade a cada mudança de estado detectada em *j*. Inicialmente, todas as entradas do vetor são iniciadas com zeros, supondo-se que todos os nós estão funcionando corretamente. Deste modo, num dado momento, um contador de eventos *par* significa que o nó pode ser considerado normal, assim como um contador de eventos *ímpar* significa que o nó pode ser considerado falho. É comparando os valores dos contadores de eventos recebidos em uma mensagem que um nó classifica a informação da mensagem como *mesma, nova,* ou *antiga*.

2) Um vetor denotado por $testb_i[1...N]$, onde $testb_i[j]$, onde $testb_i[j]$ pode assumir os valores 0,1,2,3 ou 4, dependendo de uma das seguintes situações: 0 se *i* e *j* não são vizinhos, 1 se *i* é testado por *j*, 2 se *i* testa *j*, 3 se *i* e *j* não

realizam testes um no outro, mas são vizinhos ou 4, caso *i* e *j* testem-se mutuamente.

Cada mensagem trocada entre os nós consiste do seguinte:

1) Um vetor de contadores de eventos *msg.event* [*1...N*] contendo os estados de funcionamento mais atualizados detectados sobre os nós da rede dadas as atualizações que a mensagem sofreu até o momento ao transitar pela rede.

2) Um vetor denotado por *msg.intrapath* [*1...N*], onde *msg.intrapath* [*j*] é 1 se a mensagem já foi processada pelo nó *j*, e 0 caso contrário. Esta informação é utilizada para reduzir o número de mensagens redundantes através da rede.

Utilizando estas estruturas de dados, quando uma mensagem *msg* é recebida por um nó *i*, os contadores de eventos trazidos na mensagem são comparados com os contadores de eventos mantidos no vetor $event_i$. Desta forma, uma informação sobre um nó é considerada mais recente se, e somente se, tiver um contador de eventos maior do que a entrada correspondente no vetor $event_i$. Isto é óbvio, já que o contador de eventos sempre aumenta de valor quando uma nova mudança de estado é detectada. Naturalmente, um contador menor significa uma informação mais antiga, e o mesmo contador significa a mesma informação.

O mecanismo de propagação de mensagens em paralelo, embora ofereça a melhor latência de diagnóstico possível, ocasiona o surgimento das chamadas *dead-messages*, que são mensagens redundantes e devem ser evitadas com o objetivo de diminuir o *overhead* no tráfego da rede. O vetor *msg.intrapath* é utilizado para reduzir a propagação de mensagens redundantes (*dead-messages*) através na rede. Quando um nó *i* detecta um novo evento de falha ou reparação, ele cria uma nova mensagem que será enviada para seus vizinhos, e atribui *1* à sua entrada correspondente neste vetor, isto é, atribui *1* a *msg.intrapath*[*i*]. Deste modo, quando um vizinho normal de *i* receber esta mensagem, mesmo que a reconheça como uma informação nova, não a enviará de volta para *i* - os autores chamaram este tipo de mensagem redundante de *intrapath message*. Além disso, à medida que *i* vai enviando a mensagem a seus vizinhos e recebendo suas confirmações de recebimento, *i* vai atribuindo *1* às suas entradas correspondentes, significando que

aquele vizinho já recebeu a mensagem corretamente. Usando este mecanismo, o tráfego de mensagens redundantes é reduzido, embora não seja eliminado na maioria dos casos.

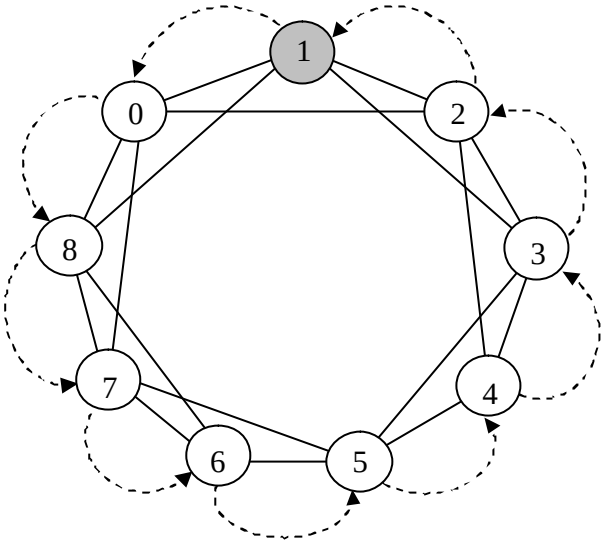


Figura 3.17 - Uma rede de nove nós e a topologia de testes

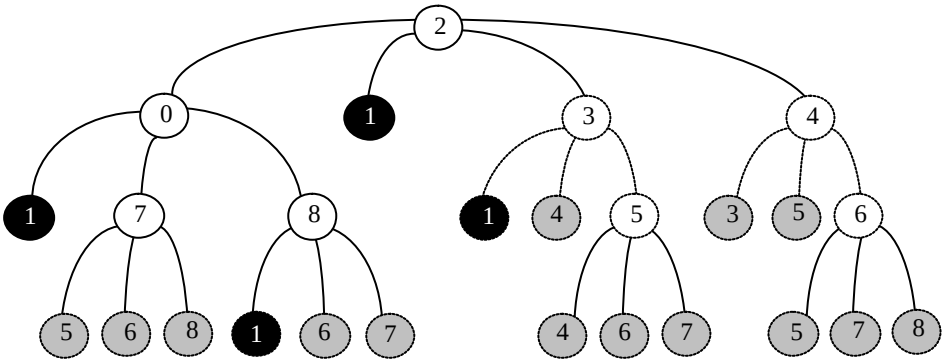


Figura 3.18 - Disseminação de mensagens em paralelo

A Figura 3.17 ilustra uma rede composta de nove nós em que o nó *1* está falho. A topologia de testes inicial está ilustrada através dos arcos tracejados.

Quando *2* detecta a falha de *1* ele inicia o processo de disseminação de mensagens em paralelo notificando este evento.

A Figura 3.18 ilustra a propagação de mensagens a partir do originador (no caso, *2*) sem nenhum mecanismo de descarte de mensagens redundantes (*dead-messages*). Neste exemplo, verificamos algumas situações particulares, por exemplo, o nó *8* envia a mensagem para o nó *7*, muito embora o seu predecessor, no caso *0*, já tivesse enviado a mesma mensagem para *7* - este é um tipo de mensagem redundante que os autores conceituaram como *first-level inter-path*. Um outro exemplo semelhante a este é visto quando o nó *5* envia a mensagem para o nó *4*, mesmo que *4* já tenha recebido a mesma mensagem de *3* anteriormente. Usando o vetor *msg.intrapath*, todas as mensagens redundantes do tipo *first-level inter-path* são eliminadas. A Figura 3.19 mostra a propagação sem estas mensagens.

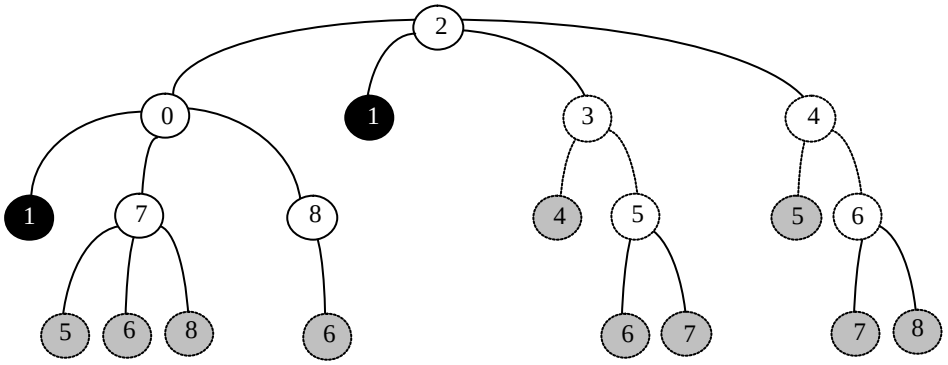


Figura 3.19 - Disseminação sem mensagens redundantes *intrapath e first-level inter-path*

Em [RAN 95] prova-se formalmente que o algoritmo proposto é correto, isto é, cada nó obtém o diagnóstico preciso dos estados de funcionamento de todos os demais, desde que o grafo original do sistema se mantenha conexo dada a situação de falha atual. Caso o grafo original se torne desconexo, ou seja, haja mais de um *componente conexo*[1] no sistema, prova-se que um nó obtém diagnóstico correto garantido apenas para todos os

[1] Subconjunto de nós normais que podem se comunicar dois a dois.

nós pertencentes ao seu próprio componente, bem como para os nós que são vizinhos dos nós deste componente (os quais são necessariamente falhos).

Devido à sua estratégia de usar uma topologia de testes ótima (cada nó é testado por um único outro), o algoritmo *RDZ* não é capaz de detectar uma situação de falha em uma configuração de nós que os autores chamaram de *jellyfish*[1] *fault node configuration*. Nesta configuração de falha, entre dois componentes conexos existe um conjunto de nós tal que parte destes nós testam-se uns aos outros de uma forma cíclica, e outros testes emanam deste ciclo. Mesmo que todos os nós da configuração *jellyfish* se tornem falhos simultaneamente, os nós dos componentes conexos não detectarão, e por conseguinte não serão capazes de diagnosticar as falhas. Esta é, sem dúvida, a maior deficiência do algoritmo *RDZ*, visto que esta configuração pode envolver um número arbitrário de nós.

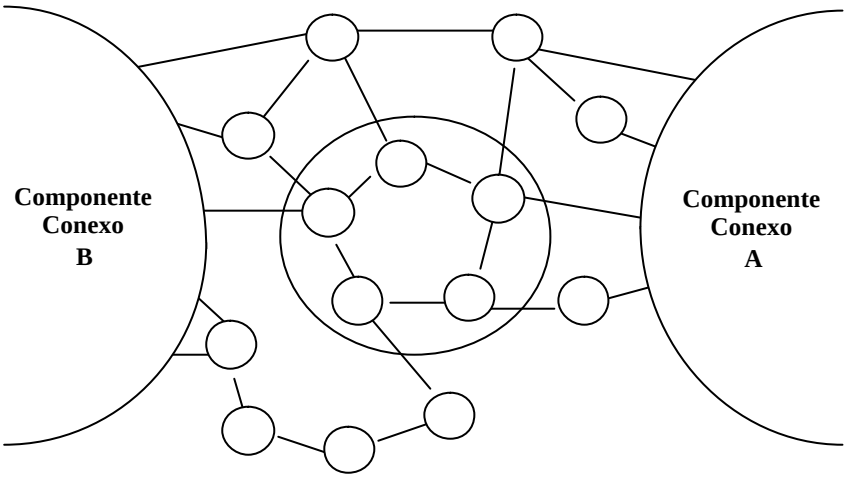


Figura 3.20 - Configuração de falhas *jellyfish*

Considere a Figura 3.20. Entre os dois componentes conexos vemos uma configuração *jellyfish*, na qual alguns nós testam-se de forma cíclica, como destacado, e outros testes partem do ciclo formando *tentáculos*. Ainda que todos estes nós que formam

[1] Água-viva. Os autores usaram esta designação por que esta configuração possui uma *cabeça*, formada por testes cíclicos, e alguns outros testes partem do ciclo formando *tentáculos*, numa forma que assemelha-se à de uma água-viva.

a configuração *jellyfish* se tornassem falhos simultaneamente, isto não seria detectado pelo algoritmo *RDZ*.

Os autores apresentaram ainda vários resultados obtidos através de simulações do algoritmo usando a linguagem CSIM. [SCH 95]

3.6.4 Algoritmo DNMN

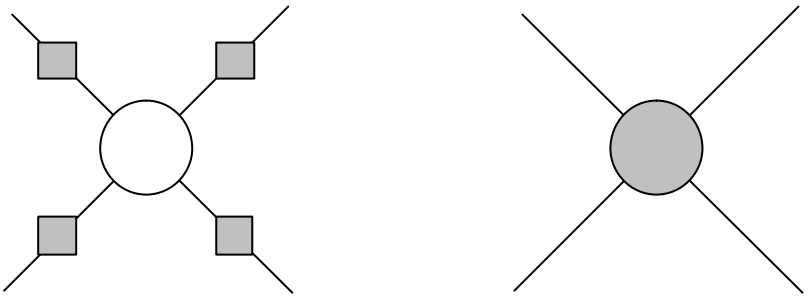
Duarte *et al.* [DUA 98b] apresentaram um algoritmo para diagnóstico distribuído em redes não *broadcast*, aplicável ao gerenciamento de falhas em redes ponto a ponto de topologia geral, o qual chamaremos de *DNMN*, das iniciais dos autores. Este algoritmo assume que cada nó conheça toda a topologia de interligação da rede, e trata de forma eficaz as falhas nos enlaces de comunicação.

O algoritmo utiliza uma estratégia de diagnóstico baseada em *time-outs* nos *enlaces de comunicação* entre os nós. Utilizando esta estratégia, se um nó *i* é responsável por realizar testes em um vizinho *j* e o último teste realizado falhou, então *i* propagará uma mensagem notificando que o enlace de comunicação entre *i* e *j* teve seu tempo de resposta ao teste excedido (*timed-out*). As mensagens de diagnóstico são propagadas em paralelo, usando a mesma abordagem apresentada em [RAN 95]. Quando um nó da rede recebe uma informação notificando o *time-out* em um enlace qualquer, ele executa um procedimento, usando a topologia de interligação, para calcular quais nós se tornaram inatingíveis. Alternativamente, ao receber uma informação notificando que um enlace foi reparado, um nó, executando o mesmo procedimento, calculará quais nós se tornaram atingíveis.

Esta abordagem de diagnóstico, baseada em *time-outs* dos enlaces de comunicação e cálculo de nós atingíveis ou inatingíveis, é considerada mais próxima da realidade do que as abordagens anteriores, visto que, de fato, é impossível distinguir a falha de um nó da falha conjunta em todos os enlaces de comunicação que levam até ele. Além do mais, em algoritmos anteriores, como por exemplo *Adapt* ou *RDZ*, dois nós em *componentes conexos* distintos podem manter estados desatualizados (incorretos) um do outro, o que não ocorre no algoritmo *DNMN*.

A Figura 3.21 ilustra duas situações em particular que reforçam a idéia de que a abordagem do algoritmo é mais realista. Na situação *A*, todos os enlaces de comunicação que levam até o nó estão falhos. Na situação *B*, o próprio nó está falho.

A partir de resultados de testes é impossível para qualquer outro nó da rede decidir qual das duas situações é a verdadeira, logo, parece bem mais razoável diagnosticar os nós como atingíveis ou inatingíveis, independente de seus estados de funcionamento reais, o que é feito no algoritmo *DNMN*.



Situação de Falha *A* Situação de Falha *B*

Figura 3.21 - Situações de falhas ambíguas

O algoritmo funciona com uma topologia de testes mínima, na qual cada enlace é testado por um único nó (o nó com identificador maior testa o enlace até o nó com identificador menor), no entanto, emprega um mecanismo adicional que os autores conceituaram como *two-way test*, no qual o nó que está no extremo oposto do enlace sendo testado é capaz de monitorar a atividade do testador. Este mecanismo funciona da seguinte forma: quando um nó *i* está testando o enlace até o nó *j* (*i* > *j*), *i* recupera o tempo local de *j*, e armazena este resultado em *j*, pressupondo-se que todos os nós possuem memória local. O nó *j*, por sua vez, mantém um limite máximo para um intervalo entre dois testes consecutivos realizados por *i* (*threshold*). Quando este limite é excedido, *j* propaga uma mensagem notificando a falha no enlace até *i*, e passa a testá-lo, até que o nó *i* responda aos testes, significando que o enlace está normal. Neste último caso, *j* pára de testar o enlace, que será novamente testado apenas por *i* garantindo a

topologia ótima de testes após a reparação. O mecanismo *two-way test* garante que uma configuração de falhas do tipo *jellyfish* [RAN 95] seja sempre detectada.

Cada nó mantém localmente um vetor com os contadores de eventos para cada enlace da rede, que são inicialmente zero, e são incrementados de um a cada novo evento detectado. Desta forma, um contador *ímpar* significa que um enlace está falho e um contador *par* significa que um enlace está normal. As mensagens de diagnóstico trafegam contendo estes contadores. Tanto a estratégia de propagação de mensagens em paralelo quanto o mecanismo de descarte de mensagens redundantes (*dead-messages*) são os mesmos propostos em [RAN 95].

A cada novo evento de falha ou reparação detectado por um nó ele executa um algoritmo para calcular que porções da rede se tornaram atingíveis ou inatingíveis dado este novo evento. Este procedimento pressupõe que cada nó mantenha localmente um mapa com toda a topologia de interligação da rede. A partir deste mapa, usando um procedimento que pode ser uma simples busca em largura no grafo de interligação, é que um nó é capaz de calcular quais outros são atingíveis ou não a partir de si próprio.

Esta é a maior deficiência do algoritmo *DNMN*, visto que não é fácil obter-se de forma automática o conhecimento de toda a topologia de interligação da rede. [MAN 96] Além do mais, em redes geograficamente distribuídas (a maior parte das redes ponto a ponto reais) que tenham sua topologia de interligação modificada com freqüência, esta tarefa se torna ainda mais difícil.

Os autores apresentaram em [DUA 98b] uma implementação do algoritmo baseada no protocolo padronizado de gerência de redes *Simple Network Management Protocol* (SNMP), e ainda propuseram uma abordagem integrada de gerenciamento de falhas em um ambiente composto por redes locais interligadas através de uma rede ponto a ponto. Nesta abordagem integrada, os autores propõem o uso de um algoritmo de diagnóstico para redes completamente conectadas, por exemplo *Hi-ADSD*, sendo executado em cada uma das redes locais, e o algoritmo *DNMN* sendo executado na rede ponto a ponto. Desta forma, cada nó da rede seria capaz de diagnosticar o estado de todo o sistema, mesmo num ambiente heterogêneo, como mostra a Figura 3.22.

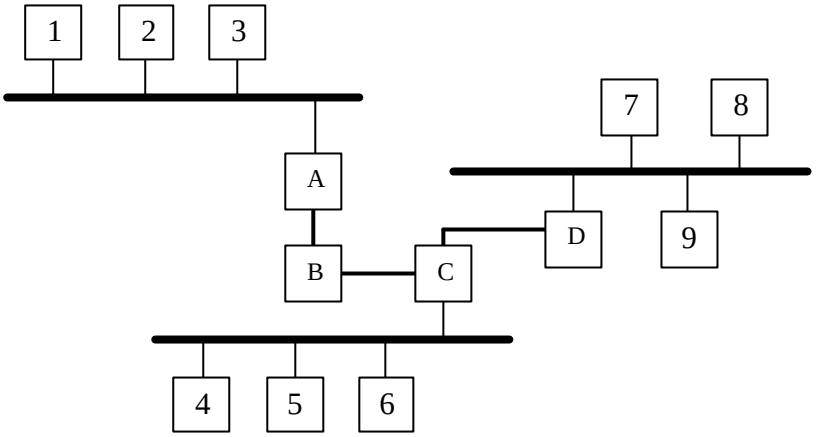


Figura 3.22 - Redes locais interligadas através de uma rede ponto a ponto

Nesta mesma figura, os nós com os identificadores de 1 a 9 estão conectados a redes *broadcast*. Os nós A, C e D estão conectados tanto a uma rede *broadcast* quanto à rede ponto a ponto que as interliga. O nó B faz parte apenas da rede ponto a ponto.

Para que um sistema de diagnóstico seja efetivo neste ambiente heterogêneo, e os dois tipos de algoritmos funcionem cooperativamente, os autores propõem que cada nó que pertença apenas a um tipo de rede execute um único algoritmo (apropriado para a rede a que estão conectados). Os nós que estão conectados aos dois tipos de topologia de interligação, por sua vez, devem executar os dois algoritmos simultaneamente, e fazer conversões das estruturas de dados de um para as estruturas de dados do outro.

CAPÍTULO 4

Um Algoritmo para Diagnóstico Distribuído de Falhas em Redes de Computadores

4.1 Introdução

Neste capítulo, apresentamos um algoritmo para *diagnóstico distribuído* de falhas em processadores (*workstations*) conectados através de uma rede de comunicação de topologia geral, que pode ser composta de enlaces ponto a ponto, canais baseados em difusão (*broadcast*) ou uma combinação arbitrária destes. Em especial, o algoritmo proposto também trata de forma eficaz a possibilidade de os enlaces de comunicação serem suscetíveis a falhas.

O algoritmo é uma evolução dos apresentados em [RAN 95] e [DUA 98b]. Como em [DUA 98b], o algoritmo trata de forma eficaz a possibilidade de falhas nos enlaces de comunicação, no entanto, como em [RAN 95], o procedimento de diagnóstico não é completo quando a situação de falha separa o sistema em subconjuntos de nós normais (*componentes normais conexos*) que não atingem uns aos outros.

Na seção 4.2, são apresentadas algumas definições que serão utilizadas ao longo do capítulo.

O algoritmo é apresentado na seção 4.3, onde mostram-se em detalhes sua especificação, funcionalidade e estruturas de dados. Alguns exemplos ilustrando a execução do algoritmo são também apresentados nesta seção.

Provamos formalmente que o algoritmo é correto na seção 4.4, dentro de condições que serão abordadas.

Alguns comentários adicionais são feitos na seção 4.5.

4.2 Definições

Definição 4.2.1 - Um *caminho* de um grafo $G = (V, E)$ é uma seqüência alternada de vértices e arestas $v_0 , e_1 , v_1 , ..., v_{n-1} , e_n , v_n$, começando e terminando com vértices, no qual cada aresta é incidente aos dois vértices que imediatamente a precedem e a seguem na seqüência. Diz-se então que v_0 *alcança* ou *atinge* v_n.

Definição 4.2.2 - Um caminho *simples* é aquele no qual todos os vértices (e assim, necessariamente todas as arestas) são distintos.

Definição 4.2.3 - Um grafo $G = (V, E)$ é *conexo* quando existir um caminho simples entre quaisquer dois nós $u \in V$ e $v \in V$. Um grafo G é dito *desconexo* se não é conexo.

Definição 4.2.4 - Um *corte de vértices* de um grafo conexo $G = (V, E)$ é um subconjunto minimal de vértices $V' \subseteq V$ cuja remoção de G o torna desconexo ou o transforma no grafo trivial (que possui um único vértice).

Definição 4.2.5 - Um *corte de arestas* de um grafo conexo $G = (V, E)$ é um subconjunto minimal de arestas $E' \subseteq E$ cuja remoção de G o torna desconexo.

Definição 4.2.6 - A *conectividade de vértices* $\kappa_v = \kappa_v(G)$ de um grafo G é a cardinalidade do menor corte de vértices de G.

Definição 4.2.7 - A *conectividade de arestas* $\kappa_a = \kappa_a(G)$ de um grafo G é a cardinalidade do menor corte de arestas de G.

Definição 4.2.8 - Denomina-se *componente conexo* de um grafo G a um subgrafo maximal de G que seja conexo. Deve ficar claro na definição que um componente conexo não é o maior subgrafo de G que é conexo, e sim, que ele não está propriamente contido em algum outro subgrafo que seja conexo.

Definição 4.2.9 - O *sistema S* é representado através de um grafo conexo $G(S) = (V(S),E(S))$, onde os processadores (*nós*) são representados pelos vértices de $V(S)$ e os enlaces são representados pelas arestas de $E(S)$, tal que $(v_x , v_y) \in E(S)$ se, e somente se, v_x e v_y são *vizinhos físicos* ou *lógicos*. Dois nós são ditos *vizinhos físicos* quando estão conectados através de um enlace ponto a ponto, e são ditos *vizinhos lógicos* quando estão

conectados através de um canal *broadcast* (ex. *Ethernet*) e foram convencionados como vizinhos no sistema *S*. Tornando mais clara esta definição, se v_x e v_y estão conectados através de um enlace ponto a ponto eles são necessariamente vizinhos físicos. Se v_x e v_y estão conectados através de um canal *broadcast*, eles podem ou não ser convencionados vizinhos lógicos, dependendo da configuração da topologia de interligação do sistema.

Apresentaremos agora algumas definições particulares para o sistema *S*, que serão feitas usando as definições sobre o grafo *G*(*S*). A partir deste ponto, usaremos a palavra *nó* no lugar de vértice, e *enlace* no lugar de aresta.

Definição 4.2.10 - Denomina-se *caminho normal* entre dois *nós normais u* e *v* do sistema *S* a um caminho simples entre *u* e *v* em *G*(*S*) tal que todos os nós e enlaces do caminho estão normais. Diz-se, neste caso, que *u alcança* ou *atinge v*.

Definição 4.2.11 - O sistema *S* é dito *normal conexo* ou simplesmente *conexo*, quando há um caminho normal entre quaisquer dois nós normais de *S*. *S* é dito *desconexo por falha*, ou simplesmente *desconexo*, se não for conexo.

Definição 4.2.12 - Um *corte de nós falhos* no sistema *S* é um subconjunto minimal de nós que *supondo-se todos falhos* tornam *S* desconexo, de forma análoga a um corte de vértices destes nós em *G*(*S*).

Definição 4.2.13 - Um *corte de enlaces falhos* no sistema *S* é um subconjunto minimal de enlaces que *supondo-se todos falhos* tornam *S* desconexo, de forma análoga a um corte de arestas destes enlaces em *G*(*S*).

Definição 4.2.14 - A *conectividade de nós normais* ou simplesmente *conectividade* do sistema *S*, κs, é o produto da cardinalidade do menor corte de nós falhos de *S* pela cardinalidade do menor corte de enlaces falhos de *S*, assim, $\kappa s = \kappa_v(G(S)) \times \kappa_a(G(S))$.

Definição 4.2.15 - Denota-se por $V_f(S) \subseteq V(S)$ o conjunto de nós falhos do sistema *S*.

Definição 4.2.16 - Denota-se por $E_f(S) \subseteq E(S)$ o conjunto de enlaces falhos do sistema *S*.

Definição 4.2.17 - Denomina-se *componente normal conexo*, ou simplesmente *componente conexo*, do sistema *S*, ao subconjunto maximal de nós normais $C \subseteq V(S)$ que

seja conexo. Em outras palavras, há um caminho normal entre quaisquer dois nós de C no subgrafo induzido pelo grafo $G(S) - V_f - E_f$.

Definição 4.2.18 - Denomina-se *vizinhança de um componente conexo C*, ao conjunto $N(C) = \{v_i \in V_f(S) \mid \exists\, v_j \in C$ para o qual $(v_i, v_j) \in E(S)\} \cup \{v_i \in V(S) \mid \forall\, v_j \in C$ se $(v_i, v_j) \in E(S)$, então $(v_i, v_j) \in E_f(S)\}$.

Esta definição nos diz que um *nó falho pertence a N*(*C*) se, e somente se, algum de seus vizinhos for um *nó de C*, independente do estado de funcionamento do enlace que o conecta a ele. Por outro lado, um *nó normal pertence a N*(*C*) se, e somente se, *todos os enlaces que o conectam aos nós de C estiverem falhos*.

4.3 Descrição do Algoritmo

4.3.1 Visão Geral

Neste algoritmo, os nós testam ou *monitoram* uns aos outros de tal forma que *cada nó é responsável por testar todos os seus vizinhos*. Os nós conhecem quem são os seus vizinhos (físicos e lógicos) através de tabelas de configurações do algoritmo.

Nós podem estar normais ou falhos. Um nó normal é aquele que executa corretamente suas tarefas computacionais, comportando-se *continuamente* como esperado. Assume-se que um nó normal é sempre capaz de realizar um procedimento que determine precisamente o estado de funcionamento de um vizinho (*teste*). Tal procedimento pode ser, por exemplo, o envio de uma mensagem ao vizinho e início de um período de espera por uma mensagem de confirmação que seja uma função da mensagem originalmente enviada. Caso esta mensagem de confirmação não seja recebida num tempo máximo admissível (*time-out*) ou seja confirmada de forma incorreta, o nó que realizou o teste pode concluir com precisão que uma das duas situações ocorre: *este vizinho está falho,* ou *o enlace que o conecta a este vizinho está falho*, muito embora não tenha como decidir de antemão qual das duas situações é a verdadeira, ou ainda se ambas ocorrem simultaneamente. Desta forma, daqui em diante, quando nos referirmos à

detecção de uma falha ou reparação em um nó, estaremos nos referindo ao resultado do teste realizado, independente de a falha ou reparação ter ocorrido no nó ou no enlace.

Assume-se ainda que os nós normais são capazes de criar e trocar mensagens entre si, e, se em princípio o sistema inteiro funciona corretamente, isto é, a rede de comunicação e todos os processadores estão em condições normais de operação, uma mensagem gerada por um nó normal é capaz de alcançar qualquer outro conectado à rede.

Alternativamente, assume-se que um nó falho não é capaz de realizar nem responder a testes (ainda que incorretamente), e também não é capaz de receber nem enviar mensagens. Em suma, pressupõe-se que um nó falho simplesmente cessa sua operação, caracterizando o que se convenciona chamar de uma *falha permanente*. Este modelo para falhas é conhecido na literatura como *fail stop model*. [SCH 83]

Quando um nó detecta que um vizinho se tornou falho ou foi reparado, ele propaga mensagens notificando este fato através de nós vizinhos, de tal forma que estas mensagens alcancem todos os nós normais e atingíveis da rede. O nó originador da mensagem a envia para todos os seus vizinhos, que por sua vez também a enviam para todos os seus vizinhos e assim por diante, de forma paralela. Este mecanismo de propagação de mensagens em paralelo é baseado no proposto em [RAN 95], e as diferenças entre um e o outro serão abordadas na seção 4.3.3.

Desta forma, a sobrecarga no tráfego da rede é mínima quando não há mudanças de estados de funcionamento dos nós, ou seja, apenas testes são realizados, e a informação sobre um novo evento é propagada tão rápido quanto possível após ser detectada.

Nenhum mecanismo de propagação de mensagens baseado em difusão (*broadcast* ou *multicasting*) é utilizado, mesmo que a rede de comunicação ofereça esta facilidade. Assim, todas as mensagens são propagadas *ponto a ponto* entre nós vizinhos, que podem ser de fato vizinhos físicos (caso estejam conectados através de um enlace *ponto a ponto*) ou vizinhos lógicos (caso estejam conectados através de um canal *broadcast*, ex. *Ethernet*).

Não é necessário haver nenhum mecanismo de sincronização da execução do algoritmo no sistema. Especificamente, não é necessário que haja um relógio global segundo o qual os testes ou mecanismos de propagação de mensagens tenham de ser feitos de forma síncrona. Na prática, as tarefas de realização de testes e propagação de mensagens podem ainda ser realizadas de forma independente (*em paralelo*) dentro de cada nó.

Os nós ou enlaces podem falhar e serem reparados a qualquer momento, sem prejuízo para a eficácia do algoritmo, o que o conceitua como *on-line*. Em outras palavras, o algoritmo é capaz de monitorar e diagnosticar dinamicamente a situação de falha da rede, sem a necessidade de eventuais interferências ou reconfigurações externas.

Um fato importante, é que um nó executando o algoritmo não precisa ter conhecimento sobre a topologia completa de interligação dos nós do sistema, mas apenas conhecer que outros nós são seus vizinhos (físicos ou lógicos).

4.3.2 Tratamento de Falhas e Reparações

Os nós detectam eventos de falha ou reparação em nós vizinhos e iniciam a propagação de mensagens através da rede com informações sobre estes eventos.

Um evento de falha é uma *transição do estado normal para falho* sofrida por um nó, e, alternativamente, um evento de reparação é uma *transição do estado falho para normal.* Um novo evento (transição) de um nó é detectado apenas ao fim do intervalo entre dois testes consecutivos realizados sobre ele. Se os resultados de dois testes consecutivos realizados sobre um nó são diferentes, há necessariamente pelo menos uma transição, isto é, um novo evento, de falha ou reparação, dependendo, naturalmente, da ordem final dos resultados dos testes.

Assim, a precisão com que novos eventos são detectados depende da periodicidade com que os testes são realizados. Por exemplo, suponha que o nó *x* acaba de testar um vizinho *y*, tendo avaliado-o como normal. Se no ínterim entre este instante e o momento em que *x* realizará um próximo teste em *y*, este sofrer um número qualquer de

transições de falha ou reparação, tal que ao final seja novamente testado e avaliado por *x* como normal, *x* não tomará conhecimento de nenhuma destas transições, e assim, obviamente, também não propagará mensagens sobre isto.

Alguns algoritmos de diagnóstico estudados na literatura, apesar de funcionarem de forma *on-line*, permitindo falhas e reparações enquanto estão sendo executados, assumem explicitamente que um nó normal não pode falhar e ser reparado no intervalo entre dois testes consecutivos realizados sobre ele, ou seja, de uma forma não detectada. Isto se deve ao fato de que as estruturas de dados deste nó específico podem ter sido reiniciadas ou não apresentarem valores consistentes imediatamente após a reparação.

Em [BIA 92] e [DUA 98] esta suposição deve-se ao fato de que quando um nó é testado e avaliado como normal, suas informações locais de diagnóstico do sistema são recuperadas e consideradas corretas pelo nó que realizou o teste. No entanto, um nó recém reparado executando estes algoritmos precisa de um tempo finito para poder adquirir novamente informações consistentes sobre o sistema. Isto poderia causar uma falha no procedimento de diagnóstico, dado que as informações consideradas corretas pelo nó que realizou o teste de fato poderiam não ser. Em [STA 92], por uma razão diferente, esta situação também causaria uma falha no procedimento de diagnóstico. O problema seria devido ao fato de que um nó recém reparado, nestas circunstâncias, manteria por tempo indeterminado dados incorretos sobre a situação de falha do sistema, já que teve suas estruturas de dados reiniciadas. O nó só recuperaria dados corretos de diagnóstico quando um novo evento de falha ou reparação fosse detectado e, neste caso, os procedimentos de atualização dos dados de diagnóstico fossem efetivados. Nesta situação, pelo menos um nó, precisamente o que falhou e foi reparado, manteria um diagnóstico incorreto do sistema por tempo indeterminado.

O algoritmo apresentado em [RAN 95] apresenta a vantagem adicional na qual um nó recém reparado sempre alerta os seus vizinhos sobre a sua reparação, e, desta forma, o fato de um nó falhar e ser reparado entre dois testes consecutivos não traz nenhum prejuízo para o funcionamento correto do algoritmo.

Baseado na estratégia apresentada em [RAN 95], o algoritmo apresentado em nosso trabalho também inclui um mecanismo para o caso de um nó normal falhar e ser

reparado no intervalo entre dois testes consecutivos realizados sobre ele. Através deste mecanismo, um nó recém reparado sempre alerta os seus vizinhos sobre a sua reparação, e, deste modo, volta a fazer parte do sistema de diagnóstico de forma correta, recuperando informações atualizadas sobre o sistema a partir de seus vizinhos. A ausência deste mecanismo, diante desta circunstância, causaria um problema semelhante ao presente no algoritmo proposto em [STA 92], ou seja, o nó recém reparado poderia manter por tempo indeterminado informações incorretas sobre a situação de falha do sistema.

É conveniente observar, entretanto, que nenhuma das transições de falha ou reparação sofridas por um nó no intervalo entre dois testes consecutivos é detectada pelo nosso algoritmo. Isto não ocorre no algoritmo apresentado em [RAN 95], pois naquele caso haveria necessariamente a geração de mensagens notificando cada nova falha ou reparação.

Cada nó mantém localmente um *contador de eventos* para cada outro nó do sistema, inclusive o seu próprio. Os eventos de falha ou reparação de um nó são contabilizados por ele e pelos demais usando estes contadores. [RAN 95] Inicialmente, os contadores em todos os nós são iniciados com *zeros* supondo que todo o sistema funciona corretamente. À medida que novos eventos de falha ou reparação ocorrem, estes contadores *são incrementados e sempre aumentam de valor*.

O contador de eventos de um nó *y* é incrementado de uma unidade em apenas duas circunstâncias: a primeira ocorre quando um novo evento de falha é detectado em *y* por algum de seus vizinhos, digamos *x*. Nesta circunstância, *x* incrementa de uma unidade o contador de eventos de *y*, *tornando-o um valor ímpar*, e envia uma mensagem contendo este novo contador a todos os seus vizinhos. A segunda circunstância ocorre quando o nó *y* recebe uma mensagem na qual o seu próprio contador de eventos é um valor ímpar. Neste caso, *y* incrementa de uma unidade o seu próprio contador, *retificando-o para um valor par*, e envia uma nova mensagem a todos os seus vizinhos contendo este contador atualizado.

Quando um nó *x* detecta um evento de reparação em um vizinho *y*, esta detecção pode ter sido devida a uma reparação do próprio nó *y* ou a uma reparação do enlace *x*-*y*.

Em qualquer das duas situações, x envia uma mensagem com os valores de todos os seus contadores de eventos para y. Se o nó *y* estava falho e foi reparado, ou se na verdade o enlace *x*-*y* é que foi reparado, tal que *y* estava inatingível por *x* antes desta reparação, então o contador de eventos de *y* na mensagem de *x* será necessariamente um valor *ímpar*. Neste caso, *y* propagará uma nova mensagem para todos os seus vizinhos, inclusive *x*, com seu próprio contador de eventos atualizado para um valor *par*. Se de fato o enlace *x*-*y* é que foi reparado, e *y* já era atingível por *x* antes desta reparação, então o contador de eventos de *y* na mensagem de *x* será necessariamente um valor *par.* Neste caso, *y* não necessitará enviar uma nova mensagem que retifique o seu próprio contador de eventos.

Usando esta estratégia de incrementar em uma unidade os contadores de eventos à medida que novos eventos de falha vão sendo detectados, ou que os nós recebam mensagens com valores ímpares para seus próprios contadores, é imediato concluir que, como um evento de falha não pode ocorrer duas ou mais vezes consecutivas em um mesmo nó, *um contador de eventos ímpar sempre significará que ele está falho*. Por outro lado, sempre que um nó é reparado ou se torna atingível, ele recebe uma mensagem de um vizinho, e pode enviar uma nova mensagem que retifique o seu contador de eventos, se for necessário. Deste modo, *um contador de eventos par sempre significará que um nó está normal.*

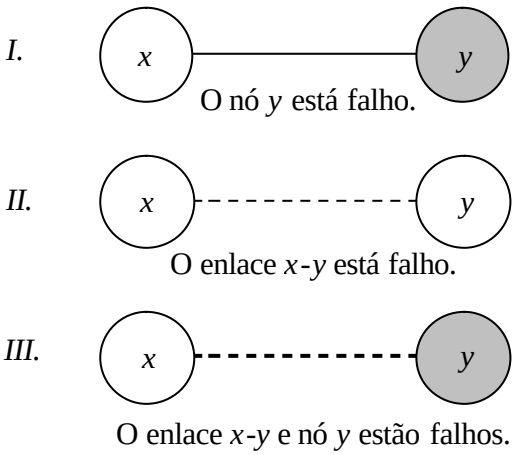


Figura 4.1 - Situações em que *x* propagará uma mensagem notificando uma falha em *y*

Se x testa o vizinho y, e o último teste falhou, tal que o resultado deste último teste é diferente do resultado anterior, x pode concluir que y está falho, ou o que o enlace x-y está falho, ou ainda que ambos estão falhos. Como não é possível saber o que ocorre de fato, em qualquer dos casos, x incrementará o contador de eventos correspondente a y de uma unidade, *tornando-o um valor ímpar, e enviará uma mensagem a seus vizinhos com este novo contador*. A Figura 4.1 ilustra as três situações.

Vale observar que, como cada nó testa todos os seus vizinhos, se apenas o enlace x-y está falho (situação *II.* da Figura 4.1), y também detectará um novo evento de falha em x e, deste modo, também propagará mensagens notificando uma falha em x.

De um modo geral, dados dois nós vizinhos x e y tal que x detectou um novo evento de falha em y, e denominando por C_x o *componente normal conexo* ao qual x pertence após este evento, as três seguintes situações devem ser consideradas:

falha-a. O nó y está falho (situação *I.* ou *III.* da Figura 4.1), neste caso, por definição, y pertence à *vizinhança* de C_x , ou, em notação, $y \in N(C_x)$

falha-b. O nó y está normal (situação *II.* da Figura 4.1), e pertence ao mesmo componente normal conexo que x ($y \in C_x$), isto é, há um *caminho normal* tal que x e y ainda são capazes de atingir um ao outro.

falha-c. O nó y está normal (situação *II.* da Figura 4.1), e pertence a um componente normal conexo disjunto de C_x, isto é $y \in C_y$ ($C_y \neq C_x$, donde $C_y \cap C_x = \varnothing$). Neste caso, não há um caminho normal tal que x e y ainda possam atingir um ao outro, e assim, por definição, $x \in N(C_y)$ e $y \in N(C_x)$.

Cada uma destas situações tem uma conseqüência diferente no funcionamento do algoritmo, vejamos:

A situação de *falha-a* causará que x propague uma mensagem notificando uma falha em y, que será eventualmente recebida por todos os demais nós de C_x. É importante observar que, além de x, pode haver outro vizinho normal de y no componente conexo C_x,

e que, se existir, também propagará mensagens notificando a falha em *y*. Este fato, no entanto, não é relevante para o funcionamento correto do algoritmo, como será visto na seção 4.3.3. Ao final do processo de disseminação das mensagens, todos os nós de C_x conhecerão corretamente o nó *y* como falho. Uma outra observação importante, é que esta falha em *y* poderá ocasionar a separação do componente normal conexo original ao qual ele pertencia, *em dois ou mais novos componentes normais conexos*. Neste caso, todos os nós de cada um deles conhecerão corretamente *y* como falho. A Figura 4.2 ilustra a circunstância em que a falha em *y* causa a separação do componente conexo original em dois novos componentes normais conexos. Neste caso, tanto o nó *x* quanto o nó *z* reconhecerão corretamente o nó *y* como falho.

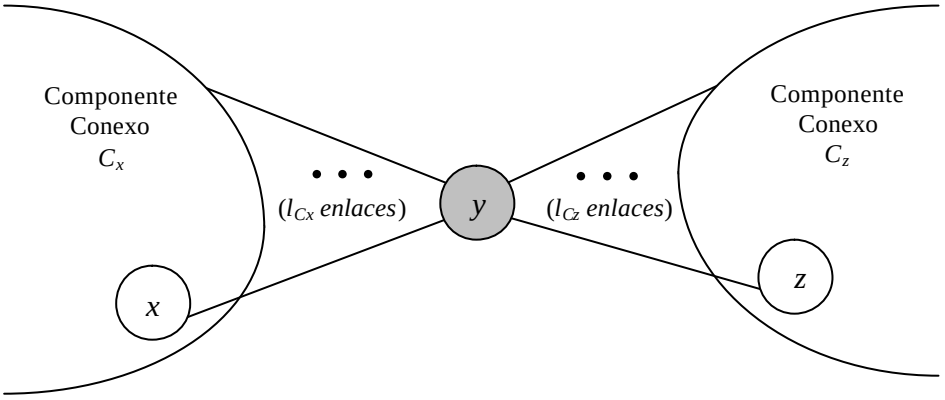


Figura 4.2 - Dois componentes normais conexos C_x e C_z após a falha do nó *y* (*falha-a*)

A situação de *falha-b* causará que duas mensagens distintas - uma notificando uma falha em *y*, e outra notificando uma falha em *x*, sejam propagadas por *x* e *y*, respectivamente, para todos os nós de C_x. Eventualmente, como ambos, *x* e *y*, pertencem ao mesmo componente normal conexo, *x* receberá a mensagem de *y* notificando a sua falha, e o mesmo ocorrerá com relação a *y* e a mensagem de *x*. Quando *x* receber a mensagem de *y* e verificar que ela trás uma informação notificando que ele próprio está falho, isto é, que o seu contador de eventos é um valor *ímpar*, ele enviará uma nova mensagem *retificando o seu estado correto*, que é normal. *Mutatis mutandis*, *y* com respeito à mensagem de *x*. Ao final do processo de disseminação de todas as mensagens,

todos os nós do componente normal conexo de x e y conhecerão corretamente ambos como normais.

A situação de *falha-c* causará que x envie uma mensagem a todos os nós de C_x e que y envie uma mensagem a todos os nós de C_y, notificando, evidentemente, a falha um do outro. Ao fim do processo de disseminação de mensagens em ambos os componentes normais conexos, x será considerado falho por todos os nós de C_y, e y será considerado falho por todos os nós de C_x, o que é correto, já que eles são inatingíveis para cada componente normal conexo. A Figura 4.3 ilustra esta situação de falha.

Figura 4.3 - Componentes normais conexos C_x e C_y gerados após a falha no enlace x-y (*falha-c*)

Cada evento de falha pode dar origem, naturalmente, a um evento de reparação correspondente. Um nó x detecta um evento de reparação em um vizinho y se ele realmente estava falho e foi reparado (*falha-a*), ou, se na verdade, o enlace x-y estava falho e foi reparado (*falha-b* ou *falha-c*). Assim, dado um evento de reparação, três situações em particular devem ser consideradas:

reparação-a. O nó y verdadeiramente estava falho e foi reparado (reparação da situação de *falha-a*).

reparação-b. O enlace x-y estava falho e foi reparado, de modo que x e y pertenciam ao mesmo componente normal conexo antes da reparação (reparação da situação de *falha-b*)

reparação-c. O enlace *x*-*y* estava falho e foi reparado, de modo que *x* e *y* pertenciam a dois componentes normais conexos distintos antes da reparação (reparação da situação de *falha-c*)

De forma análoga aos eventos de falha, cada uma das situações de reparação tem uma conseqüência diferente no funcionamento do algoritmo. Vejamos uma análise de cada uma delas:

A situação de *reparação-a* faz com que *x* envie uma mensagem para *y* contendo os valores de todos os seus contadores de eventos. Como *y* estava falho e pertencia à vizinhança de C_x antes da reparação, ele era considerado falho por todos os nós de C_x, em particular, por *x*. Deste modo, quando *y* receber a mensagem verificará que o seu próprio contador de eventos tem um valor *ímpar*. Pela funcionalidade do algoritmo, *y atualizará suas informações locais com os contadores provenientes da mensagem de x*, entretanto, *enviará uma nova mensagem a todos os seus vizinhos* com o seu próprio contador incrementado de uma unidade, *retificando-o para um valor par*. Eventualmente, esta nova mensagem percorrerá todos os nós do componente conexo C_x e todos eles atualizarão seus contadores referentes a *y* de acordo. Ao fim da propagação de todas as mensagens, *y* será conhecido corretamente como normal por todos os nós de C_x. Uma observação importante é que a reparação do nó *y* pode causar *a junção de dois ou mais componentes normais conexos disjuntos em um só*. Neste caso, após a propagação e o processamento de todas as mensagens geradas, todos os nós dos componentes normais conexos que existiam antes da reparação de *y* conhecerão os contadores de eventos mais atualizados *uns dos outros*. Deste modo, o novo componente normal conexo terá valores consistentes para todos os contadores de eventos em cada nó. A Figura 4.4 ilustra a reparação do nó *y* e a junção dos dois componentes normais conexos, para o exemplo da Figura 4.2, em um único.

A situação de *reparação-b* faz com que ambos, *x* e *y*, enviem uma mensagem um para o outro contendo, naturalmente, os seus contadores de eventos. Como eles já pertenciam ao mesmo componente conexo, os contadores respectivos um do outro já eram valores *pares*, assim, *mensagens retificadoras não serão enviadas por nenhum deles*.

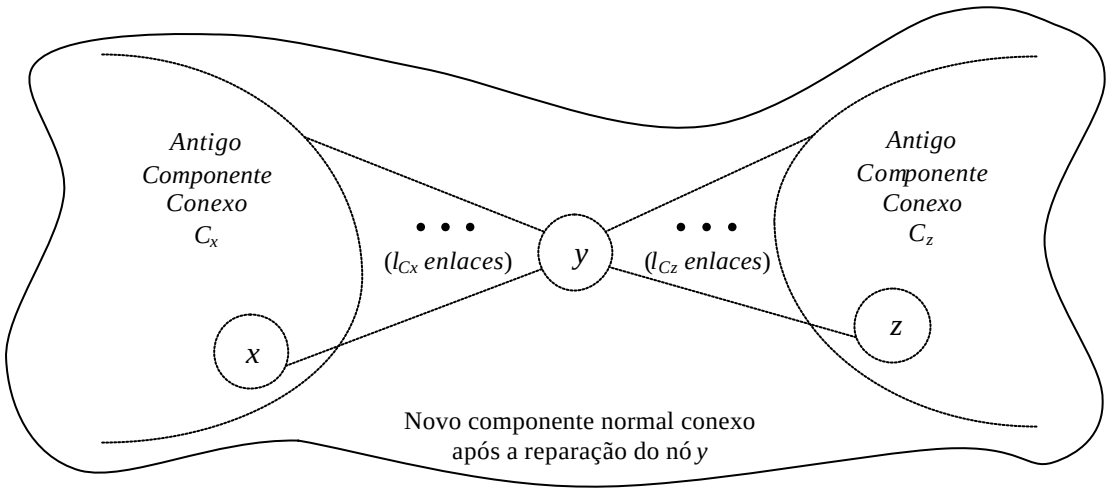


Figura 4.4 - Junção de dois componentes conexos após a reparação do nó *y* (*reparação-a*)

A situação de *reparação-c* faz com que ambos, *x* e *y*, enviem uma mensagem um para o outro contendo os seus contadores de eventos. Como *x* e *y* pertenciam a componentes conexos distintos, digamos C_x e C_y tal que *x* pertencia a $N(C_y)$ e y pertencia a $N(C_x)$, os contadores respectivos um do outro eram valores *ímpares* antes da reparação do enlace *x*-*y*. Assim, de forma análoga a situação de *reparação-a*, *x* e *y propagarão mensagens retificadoras* que serão recebidas por todos os nós do novo componente normal conexo ao qual eles agora pertencem. Este novo componente conexo é formado, naturalmente, pela *união* dos dois componentes normais conexos anteriores ($C_x \cup C_y$). Ao fim do processamento de todas as mensagens, todos os nós de $C_x \cup C_y$ conhecerão ambos, *x* e *y*, como normais. Uma observação importante, é que após o processamento de todas as mensagens, tanto os nós de C_x quanto os nós de C_y conhecerão os contadores de eventos mais atualizados *uns dos outros*. Assim, o novo componente normal conexo $C_x \cup C_y$ terá valores consistentes para todos os contadores de eventos em cada nó após a reparação do enlace *x*-*y*. A Figura 4.5 ilustra esta circunstância.

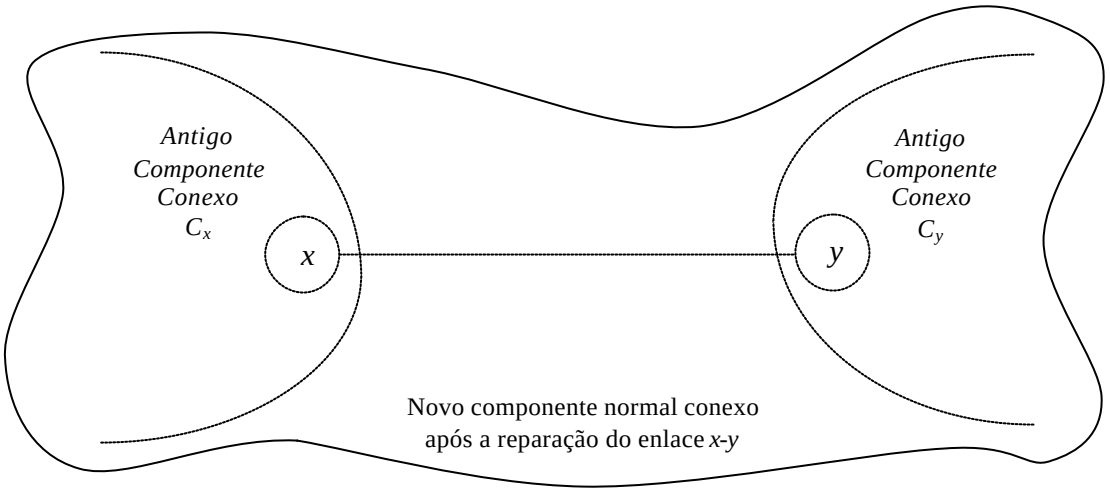


Figura 4.5 - Junção de dois componentes conexos após a reparação do enlace *x-y* (*reparação-c*)

Como dito anteriormente, se um nó falha e é reparado arbitrariamente no intervalo entre dois testes consecutivos realizados sobre ele, estas transições não são contabilizadas pelo algoritmo. Entretanto, um mecanismo adicional é incluído para evitar que este nó recém reparado mantenha dados incorretos sobre a situação de falha do sistema. Este mecanismo funciona da seguinte forma: quando um nó *y* é reparado, ele inicia seus contadores de eventos locais com *zeros* e envia uma mensagem contendo estes contadores para todos os seus vizinhos, dentre eles, digamos, *x*. Ao receber esta mensagem, *x* compara os contadores por ela trazidos com os seus próprios contadores, e, se houver pelo menos um que seja *mais atual*, ou seja, com um valor maior que zero, *x* envia uma mensagem com seus contadores de eventos de volta para *y*. Deste modo, apesar de que as transições sofridas por *y* não tenham sido percebidas pelos seus vizinhos, nem por *x*, inclusive, quando *y* é reparado ele recupera as informações mais atuais disponíveis no componente normal conexo de *x*, ao qual ele agora também pertence. Este mecanismo é o uso normal da estratégia de tratamento, dada por um nó, para as mensagens recebidas de um vizinho, como será melhor explicado na seção seguinte.

4.3.3 Estruturas de Dados e Tratamento de Mensagens

Cada nó mantém localmente um contador de eventos para cada outro nó do sistema, inclusive o seu próprio. Estes contadores de eventos são atualizados à medida que o próprio nó detecte mudanças de estado em nós vizinhos, ou ainda receba mensagens que contenham informações mais atualizadas do que as suas informações locais.

Cada mensagem que trafega pela rede carrega todos os contadores de eventos do nó que a originou. Uma mensagem pode sofrer modificações à medida que trafega pela rede.

As duas estruturas de dados mantidas localmente em cada nó x são as seguintes:

i. Um vetor com os contadores de eventos de todos os N nós da rede, denotado por $contadores\text{-}de\text{-}eventos_x$ [$1...N$], onde $contadores\text{-}de\text{-}eventos_x$ [y] contém o valor de um contador de eventos que representa o mais recente estado conhecido por x sobre y, dados os testes realizados sobre y, e as mensagens que chegaram em x até este momento. Cada entrada y deste vetor é sempre incrementada de uma unidade a cada notificação de mudança no estado de y. As mensagens que notificam mudanças no estado de y podem ter sido originadas pelos vizinhos de y ao detectarem eventos de falha nele, ou pelo próprio y, como mensagens *retificadoras*, quando ele está normal e recebe uma mensagem notificando a si próprio que está falho. Inicialmente, todas as entradas deste vetor são iniciadas com *zeros*, pressupondo-se que todos os nós funcionam corretamente. Deste modo, num dado momento, sempre ocorrerá que se $contadores\text{-}de\text{-}eventos_x$ [y] é um valor par o nó y é considerado normal por x, ao passo que se este valor é ímpar, y é considerado falho ou inatingível por x.

ii. Um vetor denotado por $vizinho_x$ [$1...N$], onde $vizinho_x$ [y] é 1 se x e y são vizinhos, e 0, caso contrário. Esta informação é dada a cada nó executando o algoritmo na forma de tabelas de configuração.

Cada mensagem trocada entre nós vizinhos consiste do seguinte:

iii. Um vetor de contadores de eventos *msg.contadores-de-eventos* [*1...N*] contendo os contadores de eventos mais atualizados disponíveis pelo nó originador da mensagem no momento em que a enviou.

iv. Um vetor denotado por *msg.nós-visitados* [*1...N*], onde *msg.nós-visitados* [*y*] é *1* se a mensagem já foi processada pelo nó *y*, e *0* caso contrário. Esta informação é utilizada para reduzir o número de mensagens redundantes trafegando na rede.

Quando um nó *x* detecta um evento de falha em um vizinho *y* ele incrementa de uma unidade o contador de eventos local correspondente a *y*, isto é, ele incrementa de uma unidade a entrada $contadores\text{-}de\text{-}eventos_x$ [*y*], *tornando-a um valor ímpar*. Em seguida, *x* compõe uma mensagem com os todos os seus contadores de eventos e a envia para todos os seus vizinhos. Para que a mensagem consiga alcançar os outros nós da rede é necessário, naturalmente, que *x* tenha pelo menos um outro vizinho que esteja normal.

Antes de enviar uma *mensagem notificando um evento*, um nó *x* sempre atribui o valor *1* à sua própria entrada e às entradas correspondentes a todos os seus vizinhos no vetor *msg.nós-visitados*. É importante ressaltar que *x* atribui o valor *1* às entradas correspondentes aos seus vizinhos antes mesmo de enviar as mensagens e ter certeza se estes vizinhos, ou os enlaces que o ligam a eles, estejam falhos ou normais. Esta estratégia é diferente da empregada em [RAN 95].

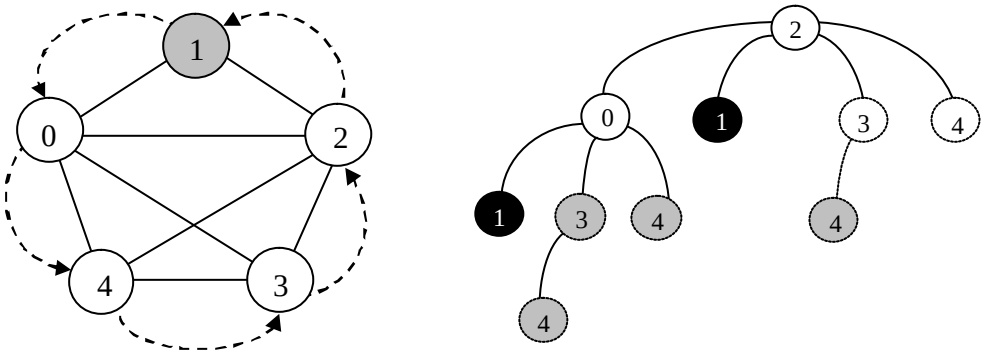


Figura 4.6 - Abordagem de propagação de mensagens empregada pelo algoritmo *RDZ*

Na abordagem apresentada em [RAN 95], *x* vai atribuindo *1* às entradas correspondentes a seus vizinhos em *msg.nós-visitados* à medida que vai enviando e recebendo confirmações de que estes vizinhos tenham recebido as mensagens com sucesso, ou identificado que acabou de enviar a mensagem para o próprio vizinho cuja falha ele está notificando. A Figura 4.6 ilustra um exemplo de propagação de mensagens em um sistema exemplo com cinco nós usando a abordagem apresentada em [RAN 95].

Na Figura 4.6 o nó *1* está falho, e os testes estão ilustrados através dos arcos tracejados. O nó *2* detecta a falha em *1* e inicia a propagação de mensagens através de seus vizinhos. Podemos observar facilmente que as mensagens enviadas pelo nó *0* para *3* e *4* e as mensagens enviadas pelo nó *3* para *4* são desnecessárias, desde que o próprio nó *2* consiga enviar estas mensagens com sucesso. A Figura 4.7 ilustra a abordagem empregada pelo nosso algoritmo.

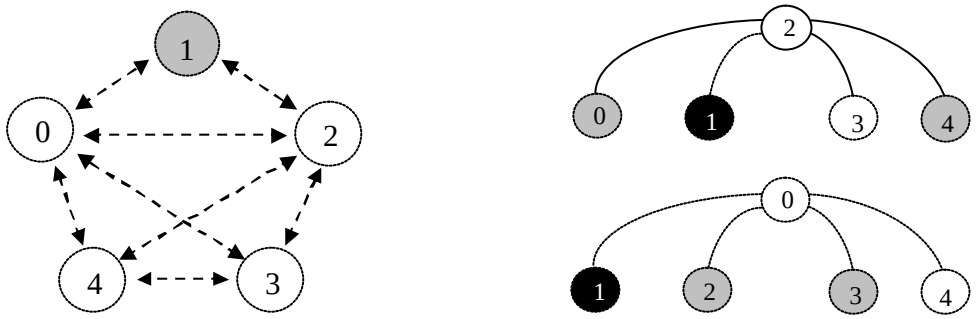


Figura 4.7 - Abordagem de propagação de mensagens empregada pelo nosso algoritmo

Na Figura 4.7 é importante observar que a topologia de testes é diferente da empregada em [RAN 95], que é uma topologia ótima na qual cada nó é testado por um único vizinho. Na topologia de testes empregada em nosso trabalho, *cada nó é testado por todos os seus vizinhos*, e, deste modo, após a falha em *1*, tanto *0* quanto *2* irão detectar e propagar mensagens notificando este evento, como está ilustrado. Se, por exemplo, tanto *0* quanto *2*, tenham detectado a falha em *1* e iniciado a propagação das mensagens simultaneamente, ambas as mensagens irão eventualmente ser recebidas por todos os demais nós deste exemplo. Em particular, a segunda mensagem que chegar será

uma mensagem redundante. Para o exemplo da Figura 4.7, as mensagens enviadas pelo nó *2* para *0* e *4* são redundantes, e o mesmo também acontece com as mensagens enviadas pelo nó *0* para *2* e *3*.

Na maioria dos casos, esta estratégia gera a propagação de mais mensagens redundantes do que a estratégia apresentada em [RAN 95], no entanto, ela impede que a falha em um enlace cause que um nó seja incorretamente considerado falho mesmo que ainda seja possível alcançá-lo por algum outro caminho. A Figura 4.8 ilustra uma situação em que o algoritmo apresentado em [RAN 95] irá cometer um erro ao diagnosticar um nó normal como falho, mesmo sendo possível alcançá-lo por um caminho alternativo. A figura ilustra uma falha no enlace *0-1* que causará que *0* propague uma mensagem notificando a falha no nó *1*. Pela estratégia de propagação de mensagens utilizada em [RAN 95], o nó *1* não será notificado de que é considerado falho, e assim, não enviará uma mensagem retificando o seu estado correto. Deste modo, o nó *1* será incorretamente considerado falho por todos os demais.

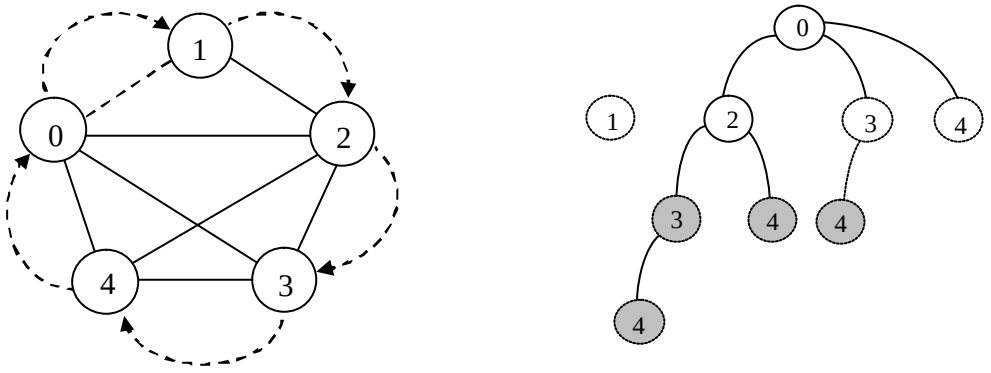


Figura 4.8 - Situação em que uma falha em enlace causará diagnóstico incorreto no algoritmo *RDZ*

Como visto na seção 4.3.2 o algoritmo proposto neste trabalho lida de forma eficaz com esta circunstância ilustrada pela Figura 4.8. Isto se deve ao uso da topologia de testes empregada, na qual cada enlace é testado de forma bidirecional, ou seja, pelos dois nós das extremidades que ele conecta. Como visto na referida seção, esta circunstância seria o que lá convencionamos chamar de uma situação de *falha-b*.

É importante ressaltar que é justamente a adoção desta topologia de testes bidirecionais o que justifica o fato do nó originador de uma mensagem poder atribuir *1* à todas as entradas correspondentes aos seus vizinhos no vetor *msg.nós-visitados*, antes mesmo de enviar as mensagens. Na prática, o único problema em potencial desta estratégia seria o fato de alguns destes vizinhos ou os enlaces que os conectam ao originador da mensagem estarem falhos. Neste caso, não seria prudente o originador afirmar de forma prévia que todos os vizinhos já tenham recebido a mensagem corretamente, pois é natural que isto poderia não ser verdade. Ora, se algum vizinho está realmente falho, não há problema algum em afirmar que ele já tenha recebido a mensagem corretamente. Muito embora isto não seja verdade, estritamente falando, também não teria maiores conseqüências. Por outro lado, se algum enlace que conecta o originador a um vizinho está falho, este vizinho também detectará uma falha no originador, e, deste modo, também propagará oportunamente uma mensagem notificando este fato, acarretando na coerência da estratégia. *Se usada no algoritmo RDZ, esta mesma estratégia seria capaz de reduzir ainda mais o número de mensagens redundantes*. Entretanto, o seu uso induziria a erro em algumas situações de falha em enlace, como ilustra o exemplo apresentado pela Figura 4.9.

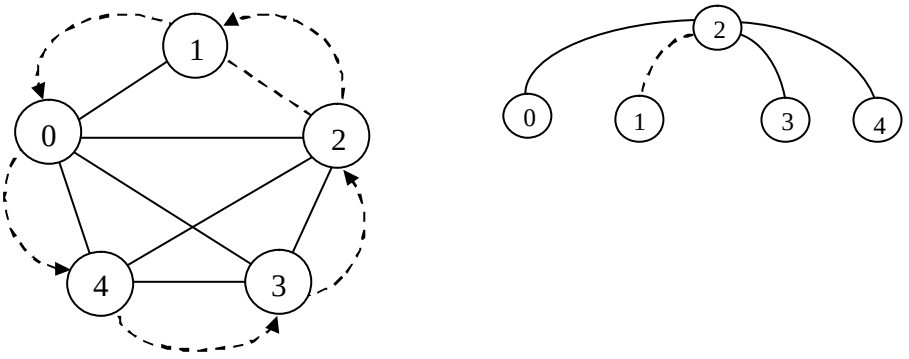


Figura 4.9 - Estratégia alternativa de propagação de mensagens induzindo a erro no algoritmo *RDZ*

No exemplo ilustrado pela Figura 4.9, o uso da nossa estratégia naquele algoritmo reduziria o número de mensagens redundantes (Figura 4.6), mas faria com que o nó *1* não

tomasse conhecimento de que fora notificado falho por *0*, e assim, também não seria motivado a enviar uma mensagem retificadora. Isto implicaria que os demais nós mantivessem, incorretamente, a informação de que o nó *1* está falho, assim como no exemplo ilustrado pela Figura 4.8. Usando a estratégia de propagação de mensagens original isto não ocorreria, como pode ser visto na Figura 4.10. Nesta mesma figura, o nó *1* receberá uma mensagem a partir do nó *0* e, desta forma, poderá enviar uma mensagem retificadora apropriadamente.

Das situações apresentadas pelas Figuras 4.8 e 4.10, decorre uma observação imediata: o algoritmo *RDZ não é necessariamente incorreto na presença de falhas em enlaces, entretanto, também não há como saber com precisão se a situação de falha em enlace levará ou não a um diagnóstico correto.* Isto se deve ao fato de que topologia de testes empregada no *RDZ*, além de mínima, é também adaptativa, modificando-se à medida que novos eventos vão sendo detectados.

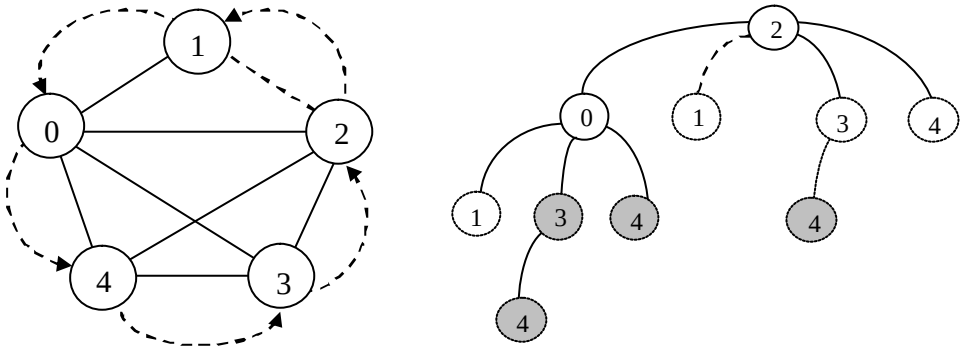


Figura 4.10 - Estratégia de propagação de mensagens empregada pelo *RDZ*

Nosso algoritmo garante um diagnóstico correto em qualquer circunstância na qual um nó normal continue atingível diante da falha em algum de seus enlaces. Isto não ocorre no algoritmo *RDZ*.

Quando um nó *k* recebe uma mensagem de um vizinho *s* ele compara os contadores de eventos trazidos na mensagem de *s* com os seus próprios contadores de eventos, e, deste modo, uma das seguintes situações pode ocorrer nesta comparação:

comparação-a. A mensagem de *s* contém os *mesmos valores para todos os contadores de eventos* que os de *k*. Neste caso, a mensagem é dita a *mesma.*

comparação-b. A mensagem de *s* contém *um valor menor para pelo menos algum dos seus contadores de eventos*, e os mesmos valores para os demais. Neste caso a mensagem é dita *antiga.*

comparação-c. A mensagem de *s* contém *um valor maior para pelo menos algum dos seus contadores de eventos* e os mesmos valores para os demais. Neste caso a mensagem é dita *nova.*

comparação-d. A mensagem de *s* é contém simultaneamente *um valor menor para pelo menos algum contador de eventos* e também *um valor maior para pelo menos algum outro contador de eventos.* Neste caso a mensagem é dita *mista*.

Sempre que um nó *k* recebe uma mensagem de um vizinho *s*, ele compara os contadores de eventos provenientes da mensagem com os seus próprios contadores de eventos mantidos localmente. Desta forma, cada uma das situações de comparação resulta em uma atitude diferente do nó *k* com relação à propagação adiante da mensagem, vejamos:

A situação de *comparação-a* causará que o nó *k* simplesmente *ignore a mensagem de s, não propagando-a adiante.*

A situação de *comparação-b causará que o nó k crie e envie uma nova mensagem, contendo todos os seus contadores de eventos, apenas para s*. Os contadores que *k* está enviando são mais atualizados que os de *s*. Antes de enviar a mensagem, *k* atribui o valor *1* à sua própria entrada e à entrada de *s* no vetor *msg.nós-visitados*.

A situação de *comparação-c causará que o nó k atualize seus contadores de eventos locais com os contadores provenientes da mensagem de s, e propague a mensagem adiante para todos os seus vizinhos que ainda não a tenham recebido.* O nó *k* sabe para quais vizinhos deve propagar a mensagem adiante observando as suas entradas correspondentes em *msg.nós-visitados*. Se algum dos vizinhos de *k* estiver com um valor *0* na sua entrada correspondente neste vetor, *k* envia a mensagem a este vizinho. É importante lembrar que, antes de enviar quaisquer mensagens, *k* atribui o valor *1* a todas as entradas com valor *0* correspondentes aos seus vizinhos em *msg.nós-visitados*.

A situação de *comparação-d causará que o nó k atualize os seus próprios contadores de eventos locais com os contadores provenientes da mensagem de s, e componha uma nova mensagem com as informações mais atualizadas provenientes tanto das informações da mensagem quanto das informações que ele já dispunha antes de recebê-la.* Feito isso, *k* envia uma nova mensagem *para todos os seus vizinhos,* atribuindo o valor *1* à sua própria entrada e às entradas correspondentes a eles.

```
01início /* no nó x */
02
03 para cada entrada n
04   contadores-de-eventos_x[n] = 0;
05
06 para cada vizinho k de x
07   envie mensagem-com-contadores-de-eventos_x para k;
08
09 repita indefinidamente
10 {
11  se existe uma mensagem na fila-de-entrada-de-mensagens então
12  {
13   receba-mensagem-do-vizinho-z;
14
15   compare (contadores-da-mensagem-de-z com contadores-de-eventos_x)
16   {
17      caso mesma-informação  /* MESMAInfo   */
18           /* A mensagem é ignorada */
19           ignore;
20
21      caso informação-antiga /* ANTIGAInfo  */
22           /* x envia uma mensagem com os seus contadores
23              de eventos de volta para z */
24           envie mensagem-com-contadores-de-eventos_x para z;
25
26      caso informação-nova   /* NOVAInfo    */
27           se contadores-de-eventos_x[x] é um valor ímpar
28           {
29              incremente de um contadores-de-eventos_x[x];
30
31              atualize contadores-de-eventos_x;
32
33              /* A propagação de uma nova mensagem é iniciada,
34                 pois x atualizou o seu próprio contador */
35
```

```
36              para cada vizinho k de x
37                   envie mensagem-com-contadores-de-eventos_x para k;
38           }
39           senão
40           {
41              atualize contadores-de-eventos_x;
42
43              /* A mensagem original deve apenas
44                 ser propagada adiante */
45
46              para cada vizinho k de x
47                   se k ainda não recebeu a mensagem original
48                      envie mensagem-original para k;
49           }
50
51      caso informação-mista   /* MISTAInfo  */
52           se contadores-de-eventos_x[x] é um valor ímpar
53              incremente de um contadores-de-eventos_x[x];
54
55           atualize contadores-de-eventos_x;
56
57           /* A propagação de uma nova mensagem é sempre iniciada,
58              pois x atualiza pelo menos um dos contadores
59              da mensagem original */
60
61           para cada vizinho k de x
62                envie mensagem-com-contadores-de-eventos_x para k;
63    }
64   }
65
66   para cada vizinho y de x
67   {
68    se intervalo-entre-testes-de-y foi excedido
69    {
70      realize-um-teste-em-y;
71
72      se evento-de-falha
73      {
74       incremente de um contadores-de-eventos_x[y];
75
76       atualize contadores-de-eventos_x;
77
78       /* x envia uma nova mensagem a todos os seus
79          vizinhos, notificando a falha em y */
80
81       para cada vizinho k de x
82            envie mensagem-com-contadores-de-eventos_x para k;
83      }
84
85      se evento-de-reparação
86
87     /* x envia uma mensagem notificando o evento
88        de reparação apenas para y */
89
90         envie mensagem-com-contadores-de-eventos_x para y;
91     }
92   }
93
94   determine o diagnóstico;
95
96 }
97
98fim.
```

Figura 4.11 - O algoritmo em pseudo código

A Figura 4.11 mostra um pseudo código para o algoritmo. No pseudo código mostrado nesta figura, é fácil verificar as estratégias de propagação de mensagens empregadas por um nó *x* ao receber uma mensagem de um vizinho *z*.

Quando *x* recebe uma mensagem de *z* contendo os mesmos valores para todos os seus contadores de eventos (**MESMAInfo**), ele simplesmente ignora a mensagem, não propagando-a adiante. Este tipo de mensagem compõe, essencialmente, o conjunto de mensagens redundantes e desnecessárias que trafegam na rede.

Uma mensagem do tipo **ANTIGAInfo** pode também ser redundante em alguns casos. Entretanto, uma mensagem atualizada é sempre enviada de volta ao vizinho do qual ela é proveniente, objetivando reduzir a *latência de diagnóstico*. Enviando uma mensagem de volta, sempre que recebe uma mensagem *antiga* de um vizinho *z*, o nó *x* faz *z* conhecer mais rapidamente as informações mais atualizadas sobre os estados das unidades do sistema que ele dispõe, contribuindo diretamente na redução da latência.

As mensagens do tipo **NOVAInfo** e **MISTAInfo** são as que contribuem efetiva e suficientemente para a notificação de um novo evento no sistema. Nenhuma destas mensagens é redundante ou desnecessária.

4.3.4 Alguns Cenários de Execução

Apresentaremos nesta seção alguns cenários com exemplos da execução do algoritmo. Todos os resultados, em cada exemplo, foram obtidos através de simulações.

No *Apêndice A* encontram-se os códigos fontes do simulador, implementado em linguagem *C*, e as respectivas simulações dos eventos de falha e reparação ilustrados pelos exemplos a seguir.

Usaremos o sistema exemplo da Figura 4.12 para ilustrar o seu comportamento e a propagação de mensagens frente a eventos de falhas e reparações em nós e enlaces. Apenas para simplificar o entendimento, consideraremos em cada exemplo a existência de um relógio imaginário segundo o qual os eventos serão referenciados. Este relógio

será iniciado de zero a partir do primeiro exemplo e prosseguirá normalmente durante os seguintes.

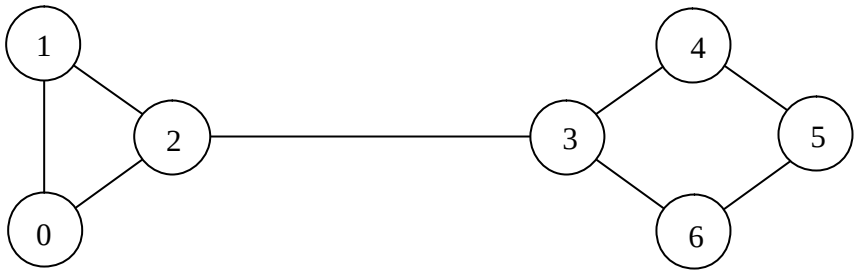


Figura 4.12 - Sistema exemplo com 7 nós e 8 enlaces funcionando normalmente

Após um evento de falha, é conveniente analisar dois casos em especial:

a. O sistema inteiro permanece *conexo*, isto é, quaisquer dois nós normais do sistema são capazes de atingir um ao outro.

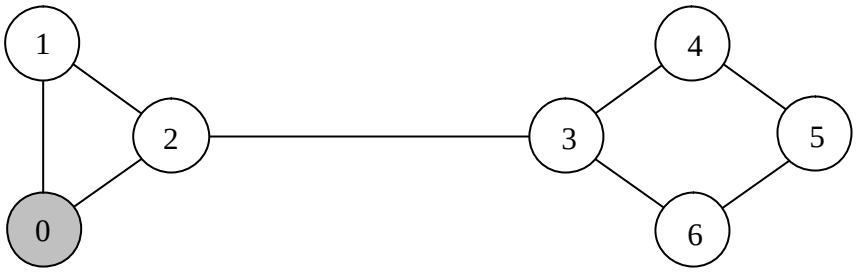
b. O sistema está *desconexo por falha*, isto é, o sistema original foi dividido em mais de um *componente normal conexo.*

Após um evento de reparação, se o sistema original se torna novamente conexo, os contadores de eventos em cada nó tornam-se consistentes, e atualizados para os estados mais recentes conhecidos uns sobre os outros.

Como primeiro exemplo, considere a Figura 4.13, que mostra uma falha no nó *0* ocorrida no tempo igual a 2 unidades, e as mensagens geradas após a detecção deste evento pelos nós *1* e *2*. Observe que esta falha não causa a desconexão dos nós normais do sistema. Do lado esquerdo estão ilustrados os testes, que são realizados regularmente neste sistema exemplo a cada 10 unidades de tempo. Do lado direito são mostradas as mensagens recebidas em cada nó, após o início da propagação das mensagens.

É importante verificar que após o processamento de todas as mensagens, cada nó normal do sistema manterá um contador de eventos de eventos igual a uma transição (1) para o nó *0*, que é um número *ímpar*, significando corretamente que *0* está falho. Observe ainda que na segunda bateria de testes (que ocorre neste exemplo ao fim das 10 primeiras

unidades de tempo), o nó *0* não mais realiza testes, já que está falho. Verificamos também que *0* não recebe nem propaga adiante nenhuma das mensagens geradas. Como visto na seção 4.3.2, este exemplo ilustra uma situação de *falha-a*.



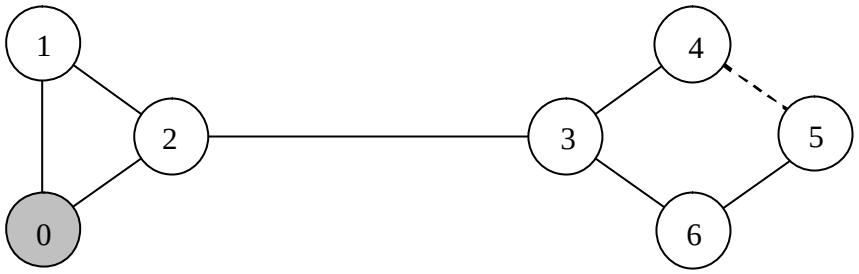

```
- nó 0 testando 1 no tempo: 0.000000
- nó 0 testando 2 no tempo: 0.000000
- nó 1 testando 0 no tempo: 0.000000
- nó 1 testando 2 no tempo: 0.000000
- nó 2 testando 0 no tempo: 0.000000
- nó 2 testando 1 no tempo: 0.000000
- nó 2 testando 3 no tempo: 0.000000
- nó 3 testando 2 no tempo: 0.000000
- nó 3 testando 4 no tempo: 0.000000
- nó 3 testando 6 no tempo: 0.000000
- nó 4 testando 3 no tempo: 0.000000
- nó 4 testando 5 no tempo: 0.000000
- nó 5 testando 4 no tempo: 0.000000
- nó 5 testando 6 no tempo: 0.000000
- nó 6 testando 3 no tempo: 0.000000
- nó 6 testando 5 no tempo: 0.000000

* falha ocorrida no nó 0 no tempo: 2.000000

- nó 1 testando 0 no tempo: 10.000000
* nó 1 testando 0 - EVENTO DE FALHA DETECTADO
- nó 1 testando 2 no tempo: 10.000000
- nó 2 testando 0 no tempo: 10.000000
* nó 2 testando 0 - EVENTO DE FALHA DETECTADO
- nó 2 testando 1 no tempo: 10.000000
- nó 2 testando 3 no tempo: 10.000000
- nó 3 testando 2 no tempo: 10.000000
- nó 3 testando 4 no tempo: 10.000000
- nó 3 testando 6 no tempo: 10.000000
- nó 4 testando 3 no tempo: 10.000000
- nó 4 testando 5 no tempo: 10.000000
- nó 5 testando 4 no tempo: 10.000000
- nó 5 testando 6 no tempo: 10.000000
- nó 6 testando 3 no tempo: 10.000000
- nó 6 testando 5 no tempo: 10.000000
```

```
MESMAInfo  recebida no nó 2 de 1 no tempo 11.000000
contadores de eventos =>  1  0  0  0  0  0  0
MESMAInfo  recebida no nó 1 de 2 no tempo 11.000000
contadores de eventos =>  1  0  0  0  0  0  0
NOVAInfo   recebida no nó 3 de 2 no tempo 11.000000
contadores de eventos =>  1  0  0  0  0  0  0
NOVAInfo   recebida no nó 4 de 3 no tempo 12.000000
contadores de eventos =>  1  0  0  0  0  0  0
NOVAInfo   recebida no nó 6 de 3 no tempo 12.000000
contadores de eventos =>  1  0  0  0  0  0  0
NOVAInfo   recebida no nó 5 de 4 no tempo 13.000000
contadores de eventos =>  1  0  0  0  0  0  0
MESMAInfo  recebida no nó 5 de 6 no tempo 13.000000
contadores de eventos =>  1  0  0  0  0  0  0
```

Figura 4.13 - Falha do nó *0* e mensagens geradas após a detecção (*falha-a*)

A Figura 4.14 mostra um novo evento de falha, desta vez ocorrido no enlace *4-5*, no tempo igual a 15 unidades de tempo. Como visto na seção 4.3.2, esta falha no enlace *4-5* corresponde a uma situação de *falha-b*. Após o processamento de todas as mensagens, cada nó manterá contadores de eventos iguais a duas transições (2) tanto para

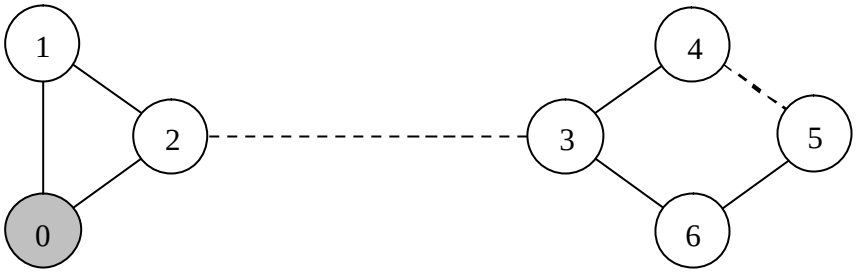
o nó *4* quanto para o nó *5*, que são números *pares*, significando corretamente que ambos estão normais.




```
* falha ocorrida no enlace 4-5 no tempo: 15.00000

- nó 1 testando 0 no tempo: 20.000000
- nó 1 testando 2 no tempo: 20.000000
- nó 2 testando 0 no tempo: 20.000000
- nó 2 testando 1 no tempo: 20.000000
- nó 2 testando 3 no tempo: 20.000000
- nó 3 testando 2 no tempo: 20.000000
- nó 3 testando 4 no tempo: 20.000000
- nó 3 testando 6 no tempo: 20.000000
- nó 4 testando 3 no tempo: 20.000000
- nó 4 testando 5 no tempo: 20.000000
* nó 4 testando 5 - EVENTO DE FALHA DETECTADO
- nó 5 testando 4 no tempo: 20.000000
* nó 5 testando 4 - EVENTO DE FALHA DETECTADO
- nó 5 testando 6 no tempo: 20.000000
- nó 6 testando 3 no tempo: 20.000000
- nó 6 testando 5 no tempo: 20.000000
```

```
NOVAInfo  recebida no nó 3 de 4 no tempo 21.000000
contadores de eventos =>  1  0  0  0  0  1  0
NOVAInfo  recebida no nó 6 de 5 no tempo 21.000000
contadores de eventos =>  1  0  0  0  1  0  0
NOVAInfo  recebida no nó 2 de 3 no tempo 22.000000
contadores de eventos =>  1  0  0  0  0  1  0
MISTAInfo  recebida no nó 6 de 3 no tempo 22.000000
contadores de eventos =>  1  0  0  0  0  1  0
MISTAInfo  recebida no nó 3 de 6 no tempo 22.000000
contadores de eventos =>  1  0  0  0  1  0  0
NOVAInfo  recebida no nó 1 de 2 no tempo 23.000000
contadores de eventos =>  1  0  0  0  0  1  0
MESMAInfo  recebida no nó 3 de 6 no tempo 23.000000
contadores de eventos =>  1  0  0  0  1  1  0
NOVAInfo  recebida no nó 5 de 6 no tempo 23.000000
contadores de eventos =>  1  0  0  0  1  1  0
NOVAInfo  recebida no nó 2 de 3 no tempo 23.000000
contadores de eventos =>  1  0  0  0  1  1  0
NOVAInfo  recebida no nó 4 de 3 no tempo 23.000000
contadores de eventos =>  1  0  0  0  1  1  0
MESMAInfo  recebida no nó 6 de 3 no tempo 23.000000
contadores de eventos =>  1  0  0  0  1  1  0
NOVAInfo  recebida no nó 6 de 5 no tempo 24.000000
contadores de eventos =>  1  0  0  0  1  2  0
NOVAInfo  recebida no nó 1 de 2 no tempo 24.000000
contadores de eventos =>  1  0  0  0  1  1  0
NOVAInfo  recebida no nó 3 de 4 no tempo 24.000000
contadores de eventos =>  1  0  0  0  2  1  0
MISTAInfo  recebida no nó 3 de 6 no tempo 25.000000
contadores de eventos =>  1  0  0  0  1  2  0
NOVAInfo  recebida no nó 2 de 3 no tempo 25.000000
contadores de eventos =>  1  0  0  0  2  1  0
MISTAInfo  recebida no nó 6 de 3 no tempo 25.000000
contadores de eventos =>  1  0  0  0  2  1  0
NOVAInfo  recebida no nó 2 de 3 no tempo 26.000000
contadores de eventos =>  1  0  0  0  2  2  0
NOVAInfo  recebida no nó 4 de 3 no tempo 26.000000
contadores de eventos =>  1  0  0  0  2  2  0
MESMAInfo  recebida no nó 6 de 3 no tempo 26.000000
contadores de eventos =>  1  0  0  0  2  2  0
NOVAInfo  recebida no nó 1 de 2 no tempo 26.000000
contadores de eventos =>  1  0  0  0  2  1  0
MESMAInfo  recebida no nó 3 de 6 no tempo 26.000000
contadores de eventos =>  1  0  0  0  2  2  0
NOVAInfo  recebida no nó 5 de 6 no tempo 26.000000
contadores de eventos =>  1  0  0  0  2  2  0
NOVAInfo  recebida no nó 1 de 2 no tempo 27.000000
contadores de eventos =>  1  0  0  0  2  2  0
```

Figura 4.14 - Falha do enlace *4-5* e mensagens geradas após a detecção (*falha-b*)

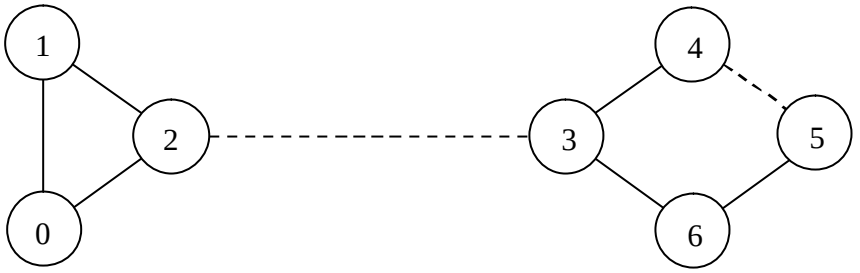



```
- nó 1 testando 0 no tempo: 30.0000 00          NOVAInfo   recebida no nó 1 de 2 no tempo 41.000000
- nó 1 testando 2 no tempo: 30.000000           contadores de eventos =>  1  0  0  1  2  2  0
- nó 2 testando 0 no tempo: 30.000000           NOVAInfo   recebida no nó 4 de 3 no tempo 41.000000
- nó 2 testando 1 no tempo: 30.000000           contadores de eventos =>  1  0  1  0  2  2  0
- nó 2 testando 3 no tempo: 30.000000           NOVAInfo   recebida no nó 6 de 3 no tempo 41.000000
- nó 3 testando 2 no tempo: 30.000000           contadores de eventos =>  1  0  1  0  2  2  0
- nó 3 testando 4 no tempo: 30.000000           NOVAInfo   recebida no nó 5 de 6 no tempo 42.000000
- nó 3 testando 6 no tempo: 30.000000           contadores de eventos =>  1  0  1  0  2  2  0
- nó 4 testando 3 no tempo: 30.000000
- nó 4 testando 5 no tempo: 30.000000
- nó 5 testando 4 no tempo: 30.000000
- nó 5 testando 6 no tempo: 30.000000
- nó 6 testando 3 no tempo: 30.000000
- nó 6 testando 5 no tempo: 30.000000

* falha ocorrida no enlace 2-3 no tempo: 35.00000

- nó 1 testando 0 no tempo: 40.000000
- nó 1 testando 2 no tempo: 40.000000
- nó 2 testando 0 no tempo: 40.000000
- nó 2 testando 1 no tempo: 40.000000
- nó 2 testando 3 no tempo: 40.000000
* nó 2 testando 3 - EVENTO DE FALHA DETECTADO
- nó 3 testando 2 no tempo: 40.000000
* nó 3 testando 2 - EVENTO DE FALHA DETECTADO
- nó 3 testando 4 no tempo: 40.000000
- nó 3 testando 6 no tempo: 40.000000
- nó 4 testando 3 no tempo: 40.000000
- nó 4 testando 5 no tempo: 40.000000
- nó 5 testando 4 no tempo: 40.000000
- nó 5 testando 6 no tempo: 40.000000
- nó 6 testando 3 no tempo: 40.000000
- nó 6 testando 5 no tempo: 40.000000
```

Figura 4.15 - Falha do enlace *2-3* e mensagens geradas após a detecção (*falha-c*)

A Figura 4.15 mostra uma falha no enlace *2-3* ocorrida a 35 unidades de tempo, que ocasionará a separação do sistema original em dois componentes normais conexos: C_2, contendo os nós *1* e *2*, e C_3, contendo os nós *3*, *4, 5* e *6*. É possível observar na figura que tanto o nó *2* quanto o nó *3* propagarão mensagens notificando a falha um do outro para cada um de seus respectivos componentes normais conexos. Deste modo, os nós de C_2 manterão um contador de eventos com valor *ímpar* para o nó *3* e, de forma análoga, os

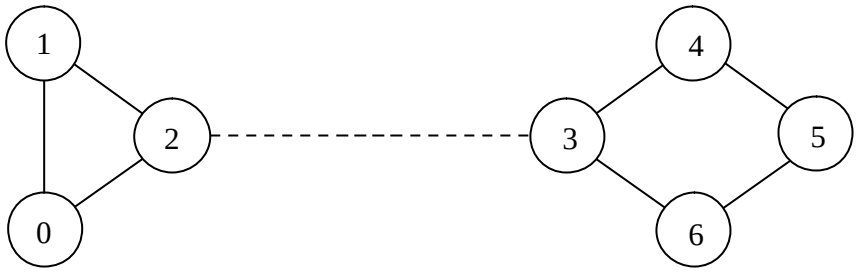
nós de C_3 manterão um contador de eventos com valor *ímpar* referente ao nó *2*. Isto é correto, já que cada nó é de fato inatingível para o componente do outro. Na seção 4.3.2 esta circunstância é que se convencionou chamar de uma situação de *falha-c*.




```
* reparação ocorrida no nó 0 no tempo: 43.00000

- nó 0 testando 1 no tempo: 50.000000
- nó 0 testando 2 no tempo: 50.000000
- nó 1 testando 0 no tempo: 50.000000
* nó 1 testando 0 - EVENTO DE REPARAÇÃO DETECTADO
- nó 1 testando 2 no tempo: 50.000000
- nó 2 testando 0 no tempo: 50.000000
* nó 2 testando 0 - EVENTO DE REPARAÇÃO DETECTADO
- nó 2 testando 1 no tempo: 50.000000
- nó 2 testando 3 no tempo: 50.000000
- nó 3 testando 2 no tempo: 50.000000
- nó 3 testando 4 no tempo: 50.000000
- nó 3 testando 6 no tempo: 50.000000
- nó 4 testando 3 no tempo: 50.000000
- nó 4 testando 5 no tempo: 50.000000
- nó 5 testando 4 no tempo: 50.000000
- nó 5 testando 6 no tempo: 50.000000
- nó 6 testando 3 no tempo: 50.000000
- nó 6 testando 5 no tempo: 50.000000

NOVAInfo  recebida no nó 0 de 1 no tempo 51.000000
contadores de eventos =>  1  0  0  1  2  2  0
ANTIGAInfo recebida no nó 0 de 2 no tempo 51.000000
contadores de eventos =>  1  0  0  1  2  2  0
NOVAInfo  recebida no nó 1 de 0 no tempo 52.000000
contadores de eventos =>  2  0  0  1  2  2  0
NOVAInfo  recebida no nó 2 de 0 no tempo 52.000000
contadores de eventos =>  2  0  0  1  2  2  0
MESMAInfo  recebida no nó 2 de 0 no tempo 52.000000
contadores de eventos =>  2  0  0  1  2  2  0
```

Figura 4.16 - Reparação do nó *0* e mensagens geradas após a detecção (*reparação-a*)

A Figura 4.16 ilustra a reparação do nó *0* ocorrida a 43 unidades de tempo. Observemos que o nó *0*, após ser reparado, agora pertence ao componente normal conexo C_2, e assim, apenas os nós deste componente serão notificados sobre sua reparação. Na seção 4.3.2 esta circunstância é o que se convencionou chamar por uma *reparação-a*. Observemos ainda que antes da reparação o nó *0* pertencia à vizinhança de C_2, ou, em notação, ao conjunto $N(C_2)$.

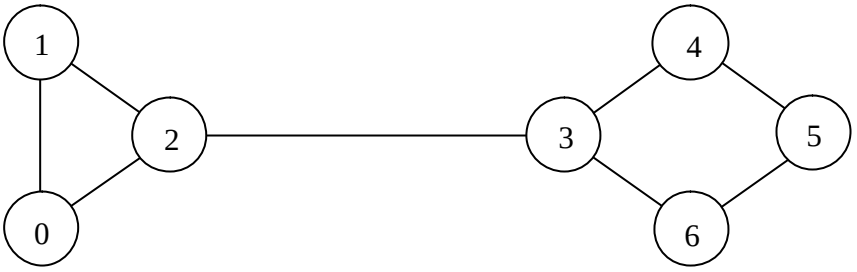



```
- nó 0 testando 1 no tempo: 60.000000
- nó 0 testando 2 no tempo: 60.000000
- nó 1 testando 0 no tempo: 60.000000
- nó 1 testando 2 no tempo: 60.000000
- nó 2 testando 0 no tempo: 60.000000
- nó 2 testando 1 no tempo: 60.000000
- nó 2 testando 3 no tempo: 60.000000
- nó 3 testando 2 no tempo: 60.000000
- nó 3 testando 4 no tempo: 60.000000
- nó 3 testando 6 no tempo: 60.000000
- nó 4 testando 3 no tempo: 60.000000
- nó 4 testando 5 no tempo: 60.000000
- nó 5 testando 4 no tempo: 60.000000
- nó 5 testando 6 no tempo: 60.000000
- nó 6 testando 3 no tempo: 60.000000
- nó 6 testando 5 no tempo: 60.000000

* reparação ocorrida no enlace 4-5 no tempo: 68.00000

- nó 0 testando 1 no tempo: 70.000000
- nó 0 testando 2 no tempo: 70.000000
- nó 1 testando 0 no tempo: 70.000000
- nó 1 testando 2 no tempo: 70.000000
- nó 2 testando 0 no tempo: 70.000000
- nó 2 testando 1 no tempo: 70.000000
- nó 2 testando 3 no tempo: 70.000000
- nó 3 testando 2 no tempo: 70.000000
- nó 3 testando 4 no tempo: 70.000000
- nó 3 testando 6 no tempo: 70.000000
- nó 4 testando 3 no tempo: 70.000000
- nó 4 testando 5 no tempo: 70.000000
* nó 4 testando 5 - EVENTO DE REPARAÇÃO DETECTADO
- nó 5 testando 4 no tempo: 70.000000
* nó 5 testando 4 - EVENTO DE REPARAÇÃO DETECTADO
- nó 5 testando 6 no tempo: 70.000000
- nó 6 testando 3 no tempo: 70.000000
- nó 6 testando 5 no tempo: 70.000000

MESMAInfo  recebida no nó 5 de 4 no tempo 71.000000
contadores de eventos =>  1  0  1  0  2  2  0
MESMAInfo  recebida no nó 4 de 5 no tempo 71.000000
contadores de eventos =>  1  0  1  0  2  2  0
```

Figura 4.17 - Reparação do enlace *4-5* e mensagens geradas após a detecção (*reparação-b*)

A Figura 4.17 ilustra a reparação do enlace *4-5* ocorrida a 68 unidades de tempo. Observemos que como *4* e *5* já pertenciam ao mesmo componente normal conexo (C_3) as mensagens que ambos enviam um para outro serão reconhecidas como a mesma informação e ignoradas. Naturalmente, nenhuma mensagem retificadora necessitará ser

enviada por nenhum deles. Esta circunstância é o que convencionamos chamar na seção 4.3.2 de uma situação de *reparação-b.*




```
* reparação ocorrida no enlace 2-3 no tempo: 77.0000

- nó 0 testando 1 no tempo: 80.000000
- nó 0 testando 2 no tempo: 80.000000
- nó 1 testando 0 no tempo: 80.000000
- nó 1 testando 2 no tempo: 80.000000
- nó 2 testando 0 no tempo: 80.000000
- nó 2 testando 1 no tempo: 80.000000
- nó 2 testando 3 no tempo: 80.000000
* nó 2 testando 3 - EVENTO DE REPARAÇÃO DETECTADO
- nó 3 testando 2 no tempo: 80.000000
* nó 3 testando 2 - EVENTO DE REPARAÇÃO DETECTADO
- nó 3 testando 4 no tempo: 80.000000
- nó 3 testando 6 no tempo: 80.000000
- nó 4 testando 3 no tempo: 80.000000
- nó 4 testando 5 no tempo: 80.000000
- nó 5 testando 4 no tempo: 80.000000
- nó 5 testando 6 no tempo: 80.000000
- nó 6 testando 3 no tempo: 80.000000
- nó 6 testando 5 no tempo: 80.000000

MISTAInfo  recebida no nó 3 de 2 no tempo 81.000000
contadores de eventos =>  2  0  0  1  2  2  0
MISTAInfo  recebida no nó 2 de 3 no tempo 81.000000
contadores de eventos =>  1  0  1  0  2  2  0
MISTAInfo  recebida no nó 2 de 3 no tempo 82.000000
contadores de eventos =>  2  0  1  2  2  2  0
NOVAInfo   recebida no nó 4 de 3 no tempo 82.000000
contadores de eventos =>  2  0  1  2  2  2  0
NOVAInfo   recebida no nó 6 de 3 no tempo 82.000000
contadores de eventos =>  2  0  1  2  2  2  0
NOVAInfo   recebida no nó 0 de 2 no tempo 82.000000
contadores de eventos =>  2  0  2  1  2  2  0
NOVAInfo   recebida no nó 1 de 2 no tempo 82.000000
contadores de eventos =>  2  0  2  1  2  2  0
MISTAInfo  recebida no nó 3 de 2 no tempo 82.000000
contadores de eventos =>  2  0  2  1  2  2  0
NOVAInfo   recebida no nó 0 de 2 no tempo 83.000000
contadores de eventos =>  2  0  2  2  2  2  0
NOVAInfo   recebida no nó 1 de 2 no tempo 83.000000
contadores de eventos =>  2  0  2  2  2  2  0
MESMAInfo  recebida no nó 3 de 2 no tempo 83.000000
contadores de eventos =>  2  0  2  2  2  2  0
NOVAInfo   recebida no nó 5 de 4 no tempo 83.000000
contadores de eventos =>  2  0  1  2  2  2  0
MESMAInfo  recebida no nó 5 de 6 no tempo 83.000000
contadores de eventos =>  2  0  1  2  2  2  0
MESMAInfo  recebida no nó 2 de 3 no tempo 83.000000
contadores de eventos =>  2  0  2  2  2  2  0
NOVAInfo   recebida no nó 4 de 3 no tempo 83.000000
contadores de eventos =>  2  0  2  2  2  2  0
NOVAInfo   recebida no nó 6 de 3 no tempo 83.000000
contadores de eventos =>  2  0  2  2  2  2  0
NOVAInfo   recebida no nó 5 de 4 no tempo 84.000000
contadores de eventos =>  2  0  2  2  2  2  0
MESMAInfo  recebida no nó 5 de 6 no tempo 84.000000
contadores de eventos =>  2  0  2  2  2  2  0
```

Figura 4.18 - Reparação do enlace *2-3* e mensagens geradas após a detecção (*reparação-c*)

Por último, o exemplo ilustrado pela Figura 4.18 mostra a reparação do enlace *2-3* ocorrida a 77 unidades de tempo. Neste caso, após esta reparação, os componentes C_2 e C_3 serão reintegrados em um único componente normal conexo, fazendo com que o sistema inteiro volte a ser conexo. Observemos que como *2* e *3* eram considerados falhos

por C_3 e C_2, respectivamente, cada um deles enviará uma mensagem retificadora corrigindo seus contadores de eventos para números *pares*. Além disso, os contadores de eventos mais atualizados disponíveis em cada um dos componentes normais conexos serão compartilhados um com o outro, à medida que as mensagens forem sendo propagadas. Assim, o evento de reparação do nó *0*, que ainda não havia sido percebido pelos nós do componente normal conexo C_3, devido exatamente à falha do enlace *2-3*, agora será conhecido corretamente por eles. Esta circunstância é o que convencionamos chamar na seção 4.3.2 de uma situação de *reparação-c*.

4.4 Prova do Corretismo[1] do Algoritmo

4.4.1 Diagnóstico Eventual

Faremos a demonstração do corretismo do algoritmo usando o conceito de *diagnóstico eventual.* [RAN 95] Neste conceito, toda a potencial complexidade que poderia advir de múltiplas falhas em nós ou enlaces, afetando a detecção de eventos e disseminação de mensagens, torna-se irrelevante, *dado que um tempo suficiente tenha decorrido desde que o último evento de falha ou reparação tenha acontecido.*

4.4.2 Prova Formal

Seja t_u o tempo no qual o último evento de falha ou reparação tenha ocorrido no sistema. Seja C um componente conexo qualquer no sistema, e $N(C)$ o conjunto de nós que formam a vizinhança de C após o tempo t_u.

Inicialmente, provaremos três *lemas* básicos sobre o algoritmo e então declararemos o seu corretismo na forma de um teorema.

[1] *Correctness proof,* na literatura técnica de língua inglesa.

Ao provar o teorema, nós teremos mostrado que num certo tempo $t_l > t_u$, cada nó pertencente ao componente normal conexo C conhecerá corretamente os estados de funcionamento para:

1. Todos os demais nós de C, os quais estão normais.
2. Todos os nós da vizinhança de C, isto é, os estados de funcionamento dos nós de $N(C)$, os quais estão falhos ou inatingíveis ao nós de C.

É importante observar que se houver mais de um componente conexo no sistema, digamos C_1 e C_2, nada se pode afirmar sobre os estados de funcionamento mantidos pelos nós de um sobre os nós do outro. Em outras palavras, dois nós normais, um pertencente a C_1 e o outro a C_2, podem manter estados de funcionamento desatualizados (incorretos) um do outro.

Para melhor entendimento das demonstrações, faremos ao longo do texto referências às linhas do algoritmo mostrado pela Figura 4.11, quando for preciso.

Lema 1: Em algum tempo $t_l > t_u$, todos os nós do componente conexo C terão os mesmos vetores de contadores de eventos, isto é, $\forall\ x, y \in C, \forall\ k, 1 \leq k \leq N$, $contadores\text{-}de\text{-}eventos_x[k] = contadores\text{-}de\text{-}eventos_y[k]$

Prova:

Considere os nós x e $y \in C$. Provaremos que x e y terão, eventualmente, os mesmos valores para todos os seus contadores de eventos, considerando os dois casos possíveis, conforme eles sejam vizinhos (caso A) ou não (caso B).

A estratégia da prova será considerar todas as possíveis combinações de falhas e reparações envolvendo os nós x e y e o enlace x-y, caso eles sejam vizinhos (caso A) e a existência de um caminho formado por nós normais vizinhos entre x e y, caso eles próprios não sejam vizinhos (caso B). Todas as demonstrações para cada caso são feitas por contradição ou usando indução matemática.

A. Os nós x e y são vizinhos em C.

A.1. O enlace x-y permaneceu normal $\forall t: 0 \leq t \leq t_u$.

A.1.1. x e y permaneceram normais $\forall t: 0 \leq t \leq t_u$. Suponha que $\exists\ k \mid$ *contadores-de-eventos*$_x[k]$ > *contadores-de-eventos*$_y[k]$, $\forall t > t_u$. Isto significa que x incrementou seu contador de eventos correspondente a k pelo menos uma vez a mais do que y, e nunca enviou uma mensagem contendo este valor para y. No entanto, pela especificação do algoritmo, quando x recebe uma nova informação que cause a atualização (incremento) do contador de eventos de algum nó do sistema, por exemplo k, ele deve enviar mensagens com este valor atualizado para todos os seus vizinhos, inclusive y (linhas 26-49 ou 51-62). Alternativamente, se o próprio x detectou um novo evento de falha em k, o que também causaria uma atualização (incremento) do seu contador de eventos, x também teria enviado uma mensagem a seus vizinhos com esta nova informação (linhas 72-83). Como y e o enlace que o conecta a x permaneceram normais $\forall t: 0 \leq t \leq t_u$, y teria necessariamente recebido a informação de x e atualizado sua entrada correspondente a k, o que é uma contradição.

A.1.2. y falhou e foi reparado pelo menos uma vez enquanto x permaneceu normal durante o período $0 \leq t \leq t_u$, isto significa que, como $y \in C$, ele deve ter realizado uma última transição de falho para normal durante o período.

A.1.2.a. x não detectou as transições de falha ou reparação sofridas por y. Neste caso, quando y for reparado pela última vez e executar o algoritmo, ele iniciará seus contadores de eventos com zeros (linhas 3-4) e enviará estes valores para todos os seus vizinhos (linhas 6-7), inclusive x. Se ao comparar a mensagem recebida de y com suas informações locais o nó x tiver pelo menos um contador mais atualizado, ele reconhecerá a informação proveniente de y como *antiga* e enviará de volta para ele os seus

contadores (linhas 21-24). Ao fim deste processo, x e y terão os mesmos valores para todos os seus contadores de eventos.

A.1.2.b. x detectou pelo menos uma transição de falha sofrida por y. Este caso será semelhante ao A.1.2.a., a única diferença é que o contador de eventos correspondente a y terá sido atualizado convenientemente para um valor ímpar. Quando ao receber a mensagem de x, y verificar que seu contador de eventos, isto é, *msg.contadores-de-eventos*[y] é ímpar, ele enviará para todos os seus vizinhos, inclusive x, uma mensagem com este valor incrementado de um, tornando-o um número par (linhas 26-49 ou 51-62). Mais uma vez, ao término deste processo, x e y terão os mesmos valores para os seus contadores de eventos.

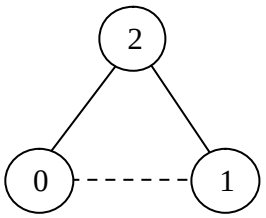
A.1.2.c. x detectou pelo menos uma transição de reparação sofrida por y. Este caso também será semelhante ao A.1.2.a, a única diferença é que um procedimento preventivo para o caso de falha no enlace x-y será executado (linhas 85-90). Assim, de forma análoga ao referido caso, e a menos do fato de que uma mensagem adicional será trocada entre eles, x e y terão os mesmos valores para os seus contadores de eventos.

A.1.3. x falhou e foi reparado pelo menos uma vez enquanto y permaneceu normal durante o período $0 \leq t \leq t_u$. Este é o mesmo caso A.1.2. para x e y com os papéis invertidos.

A.1.4. x e y falharam e foram reparados pelo menos uma vez durante o período $0 \leq t \leq t_u$. Como x e $y \in C$, ambos devem ter realizado pelo menos uma transição de falho para normal no período. Sejam t_x e t_y os instantes em que x e y tenham realizado respectivamente suas últimas transições para normais. Se $t_x < t_y$, então este é o mesmo caso A.1.2, com o período em consideração sendo $t_x \leq t \leq t_u$. De modo análogo, se $t_x > t_y$, mais uma vez este é o mesmo caso A.1.2. para x e y com os papéis invertidos.

A.2. O enlace *x*-*y* permaneceu falho a partir de um instante t_r, $\forall t: t_r \leq t \leq t_u$.

A.2.1. Suponha que *x* e *y* se mantiveram normais $\forall t: t_r \leq t \leq t_u$. Provaremos por indução matemática que após a falha no enlace *x*-*y*, todos os nós normais de *C* terão contadores de eventos iguais referentes a este evento, tanto para *x* quanto para *y*, após um tempo t_s ($t_s > t_r$). Considere um sistema composto por três nós como mostra a Figura 4.19. Suponha que o enlace entre os nós *0* e *1* está falho. Através da execução do algoritmo, *0* detectará um novo evento de falha em *1* e após incrementar o seu contador de eventos correspondente, tornando-o um valor ímpar, enviará uma mensagem a todos os seus vizinhos notificando este evento (linhas 72-83). O mesmo será feito por *1* com respeito a *0*. Após estes procedimentos, duas mensagens são geradas para este mesmo evento. Como *0* e *1* ainda pertencem ao mesmo componente conexo, e na verdade não estão falhos, eventualmente, estas mensagens informando suas respectivas falhas serão recebidas por eles. Segundo a especificação do algoritmo, se um nó *k* recebe uma mensagem *msg* de forma que *msg.contadores-de-eventos*[*k*] é um número ímpar, ele incrementa de um esta entrada, tornando-a um número par, e envia uma nova mensagem com este valor para todos os seus vizinhos (linhas 26-49 ou 51-62).




```
NOVAInfo  recebida no nó 2 de 0 no tempo 11.000000 =>  0  1  0
MISTAInfo recebida no nó 2 de 1 no tempo 11.000000 =>  1  0  0
NOVAInfo  recebida no nó 0 de 2 no tempo 12.000000 =>  1  1  0
NOVAInfo  recebida no nó 1 de 2 no tempo 12.000000 =>  1  1  0
NOVAInfo  recebida no nó 2 de 0 no tempo 13.000000 =>  2  1  0
MISTAInfo recebida no nó 2 de 1 no tempo 13.000000 =>  1  2  0
NOVAInfo  recebida no nó 0 de 2 no tempo 14.000000 =>  2  2  0
NOVAInfo  recebida no nó 1 de 2 no tempo 14.000000 =>  2  2  0
```

Figura 4.19 - Mensagens geradas para a falha do enlace *0-1*

Assim, após *0* e *1* receberem as respectivas mensagens informando suas pressupostas falhas, duas novas mensagens serão geradas retificando o fato, uma por cada um deles, de modo que eventualmente todos os nós normais do componente (*0,1* e *2*) terão entradas iguais (duas transições a mais do que os valores originais) tanto para *0* quanto para *1*. A Figura 4.19 mostra todas as mensagens geradas, de modo onde possamos comprovar que a propriedade enunciada é verdadeira para um componente composto por três nós. Suponha que esta propriedade é verdadeira para um componente conexo C que tem n elementos ($n \geq 3$). Provaremos que isto implica em que ela também seja verdadeira para um componente C' com $(n + 1)$ elementos.

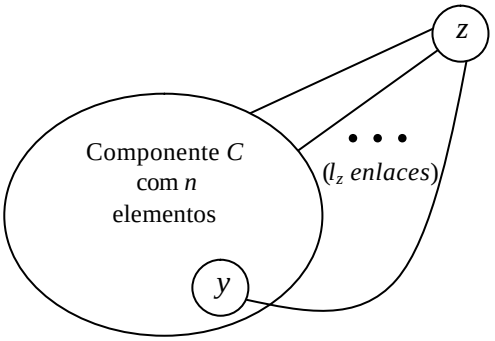


Figura 4.20 - Componente C' com $(n + 1)$ elementos

Considere a Figura 4.20. Vemos um componente C' com $(n + 1)$ elementos, de forma que o nó z possui l_z enlaces que o conectam aos nós de C, onde $1 \leq l_z \leq n$. Se ocorrer uma falha em algum enlace dentro do componente C então, por hipótese de indução, esta falha será corretamente detectada por todos os nós de C e em particular, pelo nó y. Como y e z são vizinhos, z também terá as mesmas informações de y

(caso A.1.). Se a falha ocorrer em algum dos l_z enlaces que conectam z ao componente C, por exemplo no enlace entre z e y, há de se considerar que z tenha sido escolhido inicialmente de forma que $l_z \neq 1$, pois por hipótese, z deve pertencer ao componente C'.

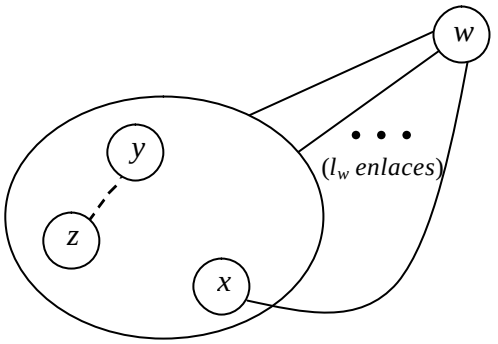


Figura 4.21 - Componente C' com $(n + 1)$ elementos visto de forma alternativa

Neste caso, basta considerar o sistema como visto na Figura 4.21. Analogamente, por hipótese de indução, a falha será detectada corretamente no componente com n elementos, e eventualmente, w terá as mesmas informações provenientes de x, que pertence a este componente.

A.2.2. Seja t_s ($t_s > t_r$), o tempo no qual todos os nós do componente teriam diagnosticado corretamente o evento de falha no enlace x-y no caso de x e y terem se mantido normais $\forall t: t_r \leq t \leq t_u$ (caso A.2.1) vejamos as seguintes situações:

A.2.2.a. x ou y falharam e foram reparados pelo menos uma vez após t_s. Neste caso, por definição, $t_s < t_u$, e como o enlace x-y permaneceu falho $\forall t: t_r \leq t \leq t_u$, basta considerar que quaisquer eventos referentes a x e y ocorridos neste período podem ser tratados como se x e y não fossem vizinhos. Isto será feito no caso B.

A.2.2.b. x ou y falharam e foram reparados pelo menos uma vez antes de t_s. Suponha que x permaneceu normal e y realizou pelo menos uma transição de falho para normal durante o período $t_r \leq t \leq t_u$.

A.2.2.b.1. Se y falhou antes de ter enviado a mensagem devido a falha no enlace x-y, então esta situação é semelhante a uma simples falha em y. Basta considerar o que ocorre a partir do momento em que há a sua última reparação. Ora, pelo caso A.1.2, y receberá de algum vizinho p os contadores de eventos mais atualizados do componente (os mesmos de x), e enviará uma nova mensagem atualizando o seu estado para normal e notificando a falha no enlace x-y. Eventualmente esta mensagem chegará em x e todos os nós do componente terão os mesmos contadores de eventos para x e y. A partir deste momento, quaisquer novos eventos com respeito a x ou y podem ser tratados como se eles não fossem mais vizinhos (caso B.)

A.2.2.b.2. Se y falhou após ter enviado a mensagem notificando a falha no enlace x-y. Esta situação é semelhante à do caso A.2.2.b.1, a única diferença é que a mensagem enviada por y terá um contador para x menor (mais antigo) ou igual em relação aos contadores correspondentes dos seus vizinhos, conforme x já tenha recebido e processado a mensagem original ou não, respectivamente. Caso o contador para x seja mais antigo, y receberá uma nova informação de algum vizinho com o contador correto (atual). A partir deste momento, quaisquer novos eventos com respeito a x ou y podem também ser tratados como se eles não fossem mais vizinhos (caso B.)

A.2.2.c. Se *y* permaneceu normal e *x* realizou pelo menos uma transição de falho para normal durante o período $t_r \leq t \leq t_u$, esta situação é semelhante a A.2.2.b para *x* e *y* com os papéis invertidos.

A.2.2.d. Se ambos, *x* e *y*, sofreram pelo menos uma transição de falho para normal cada um, no mesmo período, a situação também será a mesma vista em A.2.2.b, apenas considerando o que ocorre a partir das últimas reparações de *x* e *y* de forma análoga ao que é feito no caso A.1.4.

A.3. O enlace *x*-*y* falhou e foi reparado pelo menos uma vez no período $t_r \leq t \leq t_u$.

A.3.1. Se o enlace falhou e foi reparado durante o intervalo entre dois testes consecutivos realizados por *x* e por *y*, ou seja, de uma forma não detectada por ambos, isto não traria maiores conseqüências para o algoritmo, a menos que *x* ou *y* houvessem sofrido pelo menos uma transição de falho para normal neste período, e a mensagem notificando este fato (linhas 3-7) houvesse sido enviada no exato instante em que o enlace estivesse falho, ou ainda que uma nova informação recebida por qualquer um deles houvesse sido propagada nestas mesmas circunstâncias (linhas 26-49 ou 51-62). Este caso, no entanto, pressupõem-se impossível, pois assume-se que o mecanismo empregado para envio de mensagens seja confiável, de modo onde não ocorra o fato de uma mensagem não alcançar o seu destino sem que este isto seja notificado.

A.3.2. O enlace falhou e foi reparado pela última vez de forma que pelo menos *x* ou *y* tenha detectado este evento (linhas 72-83) e ambos permaneceram normais $\forall t: 0 \leq t \leq t_u$. Pela especificação do algoritmo, quando um nó detecta a reparação de um vizinho (linhas 85-90), ele envia uma mensagem com seus contadores de eventos para este vizinho, de forma que ele possa se reintegrar ao componente e atualizar suas informações de acordo. Do ponto de vista de um nó qualquer, a reparação de um

enlace que o conecta a um vizinho tem o mesmo significado que a reparação do próprio vizinho. Neste caso, quando o enlace em questão for reparado, x ou y enviarão uma mensagem um para o outro com seus contadores de eventos, e logo terão os mesmos valores para todos eles. Vale observar que este mecanismo para detectar a reparação de um vizinho (linhas 85-90) é, estritamente falando, desnecessário para esta tarefa, já que um nó recém reparado de fato sempre alerta sobre a sua reparação (linhas 3-7). No entanto, este mecanismo foi incluído exatamente para este caso de reparação do enlace.

A.3.3. O enlace falhou e foi reparado pela última vez de forma que pelo menos x ou y tenha detectado este evento (linhas 72-83) mas y falhou e foi reparado pelo menos uma vez enquanto x permaneceu normal no mesmo período $0 \leq t \leq t_u$. Neste caso, levando em conta o caso A.3.1, basta considerar o que ocorreu por último: a última reparação de y ou a última reparação do enlace. Em qualquer dos casos, pela especificação do algoritmo, x e y trocarão mensagens e atualizarão suas informações apropriadamente (linhas 3-7 ou 85-90).

A.3.4. O enlace falhou e foi reparado pela última vez de forma que pelo menos x ou y tenha detectado este evento (linhas 72-83) mas x falhou e foi reparado pelo menos uma vez enquanto y permaneceu normal no mesmo período $0 \leq t \leq t_u$. Este é o mesmo caso A.3.3, para x e y com os papéis invertidos.

A.3.5. O enlace falhou e foi reparado pela última vez de forma que pelo menos x ou y tenha detectado este evento (linhas 72-83) mas x e y falharam e foram reparados pelo menos uma vez no período $0 \leq t \leq t_u$. Neste caso, mais uma vez basta considerar a última das reparações de x, y ou do enlace x-y. Por A.3.3 e A.3.4, x e y terão os mesmos contadores de eventos.

B. x e y não são vizinhos em C.

Como x e y pertencem ao mesmo componente C, deve haver um *caminho normal* de tamanho t ($1 \leq t < |C|$) de x a y cruzando t enlaces normais (ou t-1 nós normais). Considere o seguinte caminho entre x e y: $x, n_1, n_2, ..., n_{t-1}, y$. Do caso A, x e n_1 possuem os mesmos contadores de eventos, desde que eles são vizinhos. De forma análoga, n_p e n_{p+1} têm os mesmos contadores de eventos ($1 \leq p \leq t$-2). Desta forma, n_{t-1} e y também terão os mesmos contadores de eventos, pois eles são vizinhos (caso A), donde conclui-se que os contadores de eventos de x e y são os mesmos. C.Q.D.

Lema 2: Em algum tempo $t_l > t_u$, todos os nós no componente conexo C terão contadores de eventos pares para todos os nós de C.

Prova:

Suponha que exista um nó y em C para o qual todos os nós de C tenham um contador de eventos *ímpar*. Desde que $y \in C$, isto implica que *contadores-de-eventos*$_y$[y] é ímpar, significando que um nó normal está mantendo a informação que ele próprio está falho. Pela especificação do algoritmo, se um nó y recebe uma mensagem *msg* de um vizinho tal que seu contador de eventos, isto é, *msg.contadores-de-eventos*[y], é ímpar, ele incrementa este valor de um, tornando-o par, e, após atualizar sua informação local, ele envia uma nova mensagem com este mesmo valor para todos os seus vizinhos (linhas 26-49 ou 51-62), informando-os que de fato está normal. Logo, por contradição, os vizinhos de y terão contadores de eventos com valores *pares* para ele. Pelo *Lema 1*, em algum tempo $t_l > t_u$, todos os nós do componente C terão os mesmos valores para os contadores de eventos de C. Isto significa que todos os nós de C terão necessariamente contadores de eventos pares para y. C.Q.D.

Lema 3: Em algum tempo $t_l > t_u$, todos os nós no componente conexo C terão contadores de eventos ímpares para todos os nós de $N(C)$.

Prova:

Suponha que exista um nó *w* em *N(C)* para o qual todos os nós de *C* tenham um contador de eventos *par*.

Inicialmente, devemos considerar que há apenas dois casos possíveis para que *w* pertença a *N(C)*. O primeiro ocorre quando o nó *w* está falho, neste caso, como *C* é composto apenas por nós normais, por definição $w \in N(C)$. O segundo caso ocorre quando *w* não está falho, mas, na verdade, todos os enlaces que o conectam a *C* estão falhos. A Figura 4.22(a) ilustra o primeiro caso e 4.22(b) o segundo.

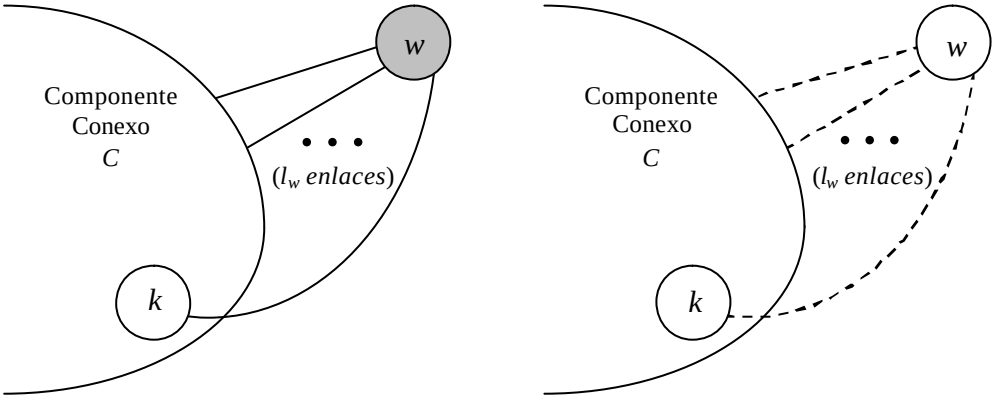


Figura 4.22 - (a) O nó *w* está falho **(b) Todos os enlaces de *w* até *C* estão falhos**

Ora, pela especificação do algoritmo, eventualmente algum vizinho de *w*, digamos *k*, detectará este evento de falha (linhas 72-83). Isto significa que *k* incrementará o contador de eventos de *w*, tornando-o um valor ímpar, e uma nova mensagem com este valor ímpar será enviada a todos os vizinhos de *k*. Se existir algum outro vizinho de *w* em *C*, digamos *r*, e *r* fizer o mesmo com respeito a *w*, antes que tenha recebido a informação propagada originalmente por *k* sobre esta mesma falha, isto não trará nenhum problema, dado que pelo *Lema 2*, como *k* e *r* pertencem a *C*, eles tinham o mesmo contador de eventos para *w* antes de sua falha. Pelo *Lema 1*, eventualmente todos os nós de *C* terão o mesmo contador de eventos para *w* que o de *k*, que é um valor ímpar. Como *w* está falho

ou inatingível, ele não receberá uma mensagem com seu contador de eventos ímpar, e deste modo não gerará uma nova mensagem com um contador par (linhas 26-49 ou 51-62). Assim, por contradição, o contador de eventos para $w \in N(C)$ será necessariamente um valor ímpar. C.Q.D.

Teorema 1: Cada nó de C conhece corretamente os estados de funcionamento dos demais nós de C, e também conhece corretamente os estados de funcionamento de todos os nós de $N(C)$, o quais estão falhos ou inatingíveis.

Prova:

Seja $x \in C$ um nó qualquer deste componente normal conexo. Pelo *Lema 2*, *x tem contadores de eventos pares para todos os nós de C*, o que, pela especificação do algoritmo, *significa que estes nós estão normais*. Alternativamente, pelo *Lema 3*, *x tem contadores de eventos ímpares para todos os nós de N(C)*, o que, também pela especificação do algoritmo, *significa que estes nós estão falhos ou inatingíveis a partir de x*. Pelo *Lema 1*, todos os demais nós de C têm todos os contadores de eventos iguais aos de x, implicando que todos os contadores são consistentes em cada nó de C. C.Q.D.

4.5 Comentários

Como visto nos exemplos mostrados pelas Figuras 4.13 a 4.18, algumas situações de falha podem causar a separação do sistema original em mais de um componente normal conexo.

Em particular, se considerarmos os exemplos mostrados pelas Figuras 4.15 e 4.16 veremos que o evento de reparação ocorrido no nó *0* (Figura 4.16) não foi percebido por um dos componentes normais conexos existente a partir da falha no enlace *2-3* (Figura 4.15).

Este tipo de problema é o mesmo que ocorre nos algoritmos apresentados em [RAN 95] e [STA 92]. À exemplo destes, caso haja dois componentes normais conexos

disjuntos, digamos C_x e C_y, o algoritmo apresentado em nosso trabalho não garante que os nós de C_x tenham contadores de eventos corretos (atualizados) sobre os nós de C_y, e o mesmo acontece com a recíproca.

Como cada nó conhece apenas quem são os seus vizinhos, ele não tem como saber se há mais de um componente normal conexo no sistema. Este fato traz uma implicação imediata para o procedimento de diagnóstico: *um nó não pode saber com precisão para quais outros nós ele tem contadores de eventos corretos (atualizados), e assim, não pode saber realmente quais outros nós estão normais ou falhos.*

O algoritmo apresentado em [DUA 98b] contorna este problema mantendo em cada nó a topologia completa de interligação da rede. No algoritmo proposto em [DUA 98b] também são usados contadores de eventos com a mesma significação relativa à paridade - *par* significa normal e *ímpar* significa falho. Os contadores, entretanto, são referentes aos enlaces e não aos nós, como proposto originalmente em [RAN 95]. Quando um nó recebe uma mensagem com o valor ímpar para o contador de eventos de um dado enlace, ele executa um procedimento para *calcular que nós se tornaram inatingíveis* dado que este enlace (ou na verdade um dos nós que ele conecta) esteja falho. Alternativamente, quando um nó recebe uma informação notificando que um enlace foi reparado, ele executa o mesmo procedimento para *calcular que outros nós se tornaram atingíveis*. Esta estratégia impede que dois nós pertencentes a componentes normais conexos distintos mantenham estados de funcionamento desatualizados um do outro, e que, naturalmente, podem não mais corresponder à realidade.

O principal problema desta estratégia é a necessidade de cada nó ter de conhecer toda a topologia de interligação da rede. É natural que esta pode ser uma tarefa bastante complexa em uma rede geograficamente distribuída, cuja topologia de interligação seja alterada com freqüência.

Diante destes fatores, o algoritmo apresentado em nosso trabalho torna-se mais interessante para sistemas com alta *conectividade de nós normais*, nos quais a probabilidade de uma situação de falha gerar mais de um componente normal conexo é realmente pequena.

CAPÍTULO 5

Simulações e Desempenho

5.1 Introdução

Neste capítulo, apresentaremos alguns resultados da execução do algoritmo, obtidos através de simulação. Serão apresentados os resultados obtidos para *tráfego de mensagens*, frente a eventos de falhas e reparações em nós e enlaces, considerando três tipos de topologia de interligação da rede: redes de conexão mínima, redes completamente conectadas e redes de topologia geral.

Inicialmente, faremos na seção 5.2 uma exposição sobre o simulador que foi desenvolvido usando o conjunto de bibliotecas para simulação de eventos discretos SMPL.

Na seção 5.3, mostraremos os gráficos com os sumários dos resultados obtidos em função dos tipos de topologia de interligação.

Na seção 5.4, apresentaremos tabelas comparativas do desempenho do nosso algoritmo com os de outros algoritmo de diagnóstico similares, estudados na literatura.

5.2 Modelagem e Implementação do Simulador em SMPL

5.2.1 Simulações de Eventos Discretos e Biblioteca SMPL

A biblioteca SMPL [MAC 80,87] oferece um conjunto de ferramentas práticas para a simulação de sistemas de computação baseados em eventos discretos. Segundo este paradigma, o funcionamento do sistema é modelado através de uma lista de eventos, representando circunstâncias instantâneas que podem ou devem ocorrer no sistema real.

A partir desta lista, cada etapa da simulação é feita usando um mecanismo que cause a ocorrência de um próximo evento. Ao ocorrer o evento, o simulador é capaz de identificá-lo e determinar qual o procedimento que deve executar para tratá-lo. Naturalmente, a lógica do procedimento para tratar um evento depende intrinsecamente da funcionalidade do sistema sendo simulado.

A biblioteca SMPL oferece um *framework* prático para a manipulação da lista de eventos, permitindo, por exemplo, escalonar novos eventos e causar a ocorrência dos mesmos. Outras rotinas úteis para simulações, como, por exemplo, rotinas geradoras de números aleatórios e de distribuições estatísticas, completam o conjunto de bibliotecas para simulação SMPL.

Este conjunto de bibliotecas foi originalmente implementado em *PL/1* por M. H. MacDougall [MAC 80], e posteriormente convertido para a linguagem *C* por Teemu Kerola. [KER 88] Nós adotamos este último conjunto em *C* para a implementação prática de um simulador do nosso algoritmo.

5.2.2 Implementação do Simulador

Nós e enlaces foram modelados em SMPL como *facilities* [KER 88], e seis eventos que compõem a funcionalidade do sistema e do algoritmo foram definidos:

- TESTE
- RECEBER_MENSAGEM
- FALHA_DE_NÓ
- REPARAÇÃO_DE_NÓ
- FALHA_DE_ENLACE
- REPARAÇÃO_DE_ENLACE

Os *eventos de falha e reparação* são simulados usando as funções *request* e *release*, respectivamente. A função *request* ocasiona uma solicitação da *facility* mudando o seu estado para *busy* (ocupado). Por outro lado, a função *release* libera uma *facility*

mudando o seu estado para *free* (livre). Em nossa implementação, o estado *busy* significa falho e *free* significa normal para uma *facility* (nó ou enlace). A função *status* é utilizada para saber o estado atual de uma *facility* (nó ou enlace). O teste de um nó *x* sobre um vizinho *y* é uma composição dos estados (*status*) de *y* e do enlace *x*-*y*. Em outras palavras, se pelo menos um deles (*y* ou o enlace *x*-*y*) estiver *busy* (falho) o teste realizado por *x* sobre *y* falhará.

Os eventos de TESTE são escalonados para ocorrerem em cada nó a intervalos regulares, de modo que, se um novo evento de falha ou reparação é detectado, os eventos de RECEBER_MENSAGEM notificando este fato, são escalonados para ocorrerem nos vizinhos correspondentes do nó que o detectou. Ao executar um evento de RECEBER_MENSAGEM, um nó compara as informações da mensagem com sua informação local e pode escalonar novos eventos RECEBER_MENSAGEM de acordo com a funcionalidade do algoritmo, como visto no *Capítulo 4*.

O *Apêndice A* contém os códigos fontes do simulador implementado em linguagem *ANSI C* usando o conjunto de bibliotecas SMPL. Nos códigos fontes estão comentadas em detalhes as estruturas de dados criadas para dar suporte à simulação.

5.3 Simulações de Execução

A seguir, mostramos os gráficos obtidos para os números médios de mensagens geradas diante de falhas e reparações de nós e enlaces. Todos os gráficos ilustram os resultados obtidos para uma falha e reparação de um nó ou enlace arbitrário em redes variando de 5 a 100 nós. Consideramos três tipos de topologias de rede usuais: *redes de conexão mínima, redes completamente conectadas e redes de topologia geral.*

Redes de conexão mínima são aquelas onde há um número mínimo de enlaces conectando os nós. Geralmente o número de enlaces é igual ao número de nós menos um, ou, no máximo, o número de enlaces é igual ao número de nós. Um *anel* é um exemplo de grafo que representa uma rede de conexão mínima.

Redes completamente conectadas são aquelas em que quaisquer dois nós são vizinhos, ou seja, quaisquer dois nós estão ligados através de um enlace, seja ponto a

ponto ou *broadcast*, e são vizinhos físicos ou lógicos. Um *grafo completo* é um exemplo de grafo que representa uma rede completamente conectada.

Redes de topologia geral têm, geralmente, uma topologia intermediária entre a de conexão mínima e a completamente conectada.

Para cada um desses três tipos de redes, a construção da topologia de conexão, que decide quais nós serão vizinhos, é criada em nosso simulador aleatoriamente. Para cada tamanho de rede (número de nós) foram analisados os números de mensagens gerados em cada um dos tipos de topologia citados anteriormente, escolhendo arbitrariamente um nó ou enlace e simulando um evento de falha e reparação.

As Figuras 5.1 a 5.4 mostram os números médios obtidos para cada tipo de mensagem diante de falhas e reparações de nós arbitrários. A Figura 5.5 mostra o total médio de mensagens para falhas e reparações de nós.

De forma análoga, as Figuras 5.6 a 5.9 mostram os resultados médios obtidos para falhas e reparações de enlaces arbitrários. A Figura 5.10 mostra o total médio de mensagens para falhas e reparações de enlaces.

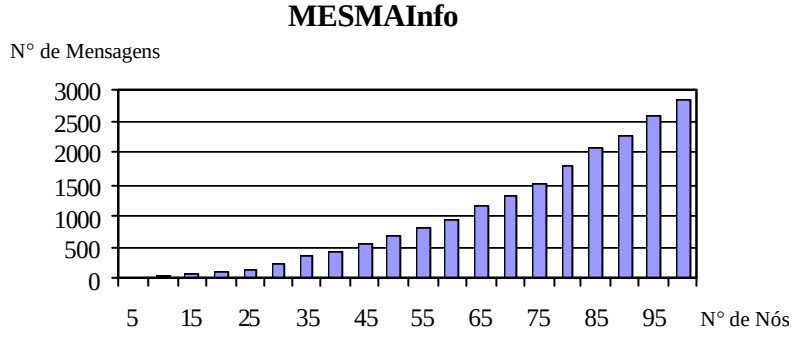


Figura 5.1 - Números médios de mensagens MESMAInfo para a falha e reparação de um nó

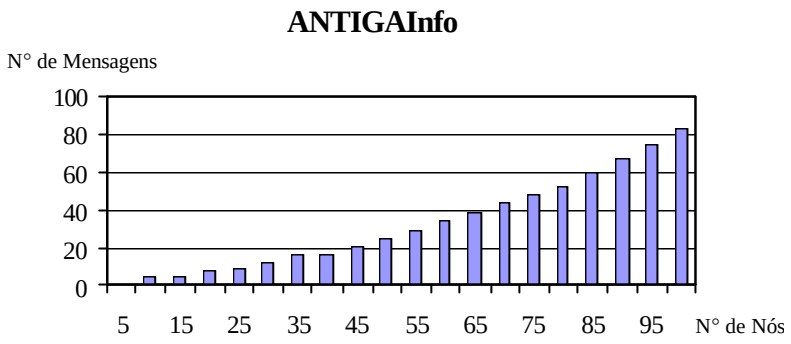


Figura 5.2 - Números médios de mensagens ANTIGAInfo para a falha e reparação de um nó

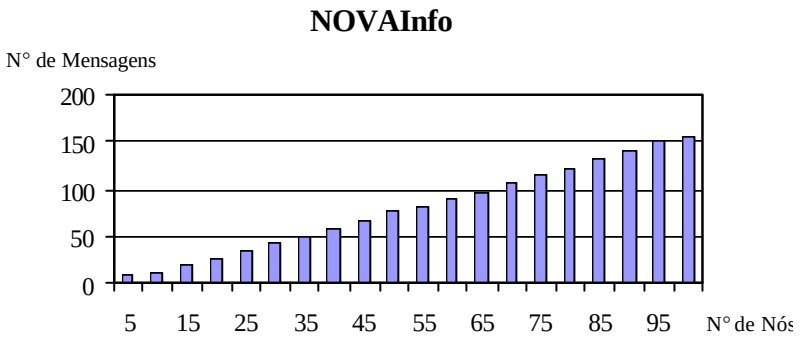


Figura 5.3 - Números médios de mensagens NOVAInfo para a falha e reparação de um nó

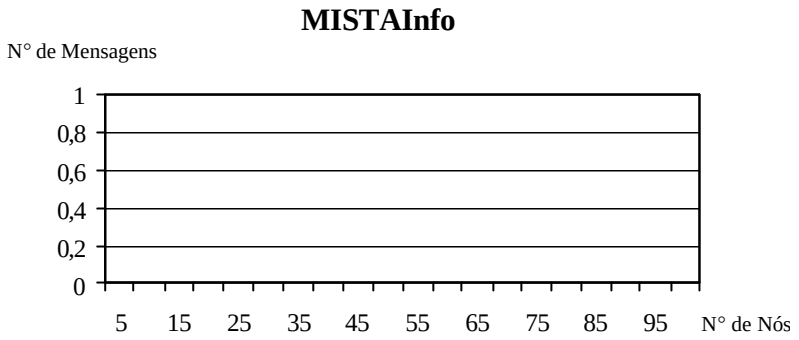


Figura 5.4 - Números médios de mensagens MISTAInfo para a falha e reparação de um nó

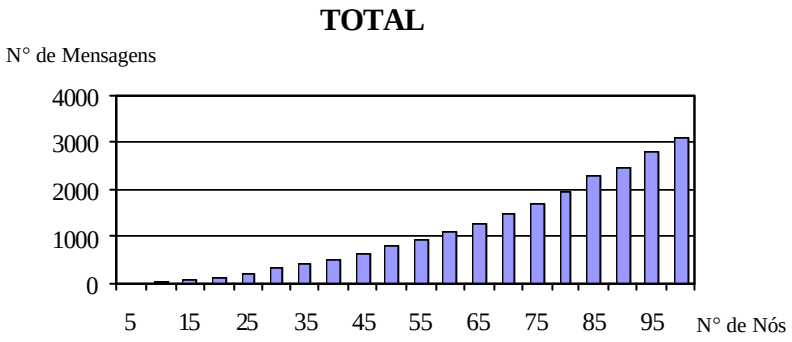


Figura 5.5 - Números médios totais de mensagens para a falha e reparação de um nó

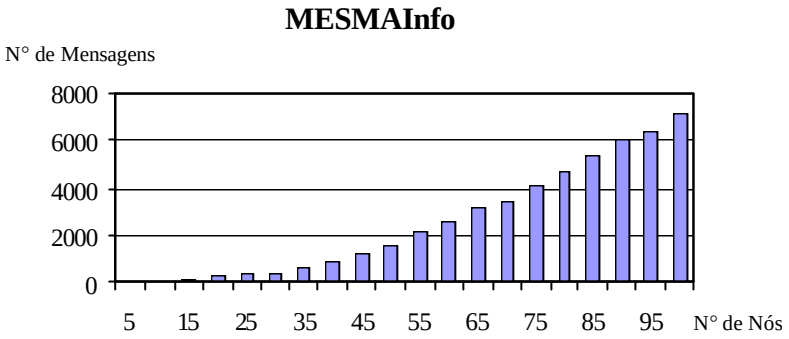


Figura 5.6 - Números médios de mensagens MESMAInfo para a falha e reparação de um enlace

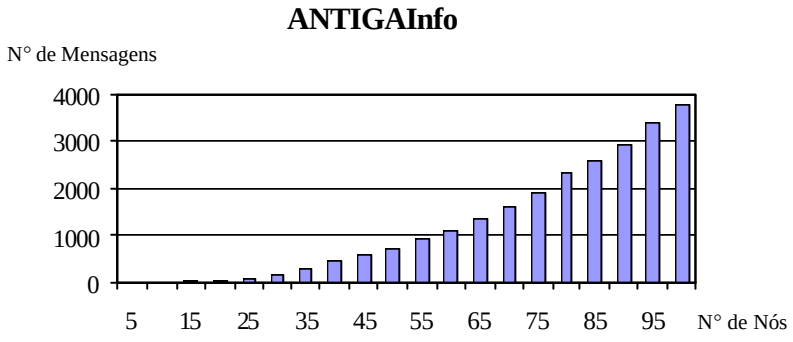


Figura 5.7 - Números médios de mensagens ANTIGAInfo para a falha e reparação de um enlace

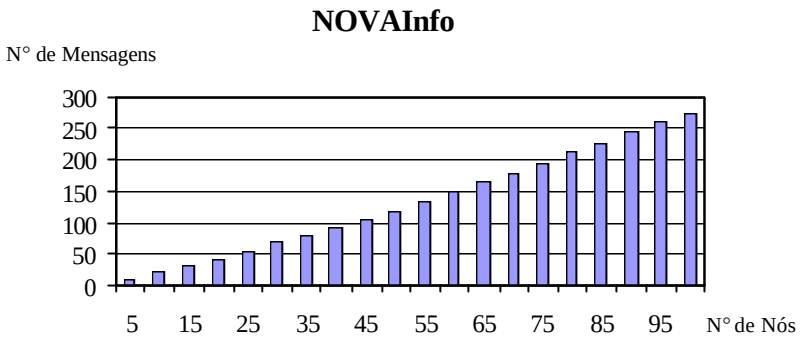


Figura 5.8 - Números médios de mensagens NOVAInfo para a falha e reparação de um enlace

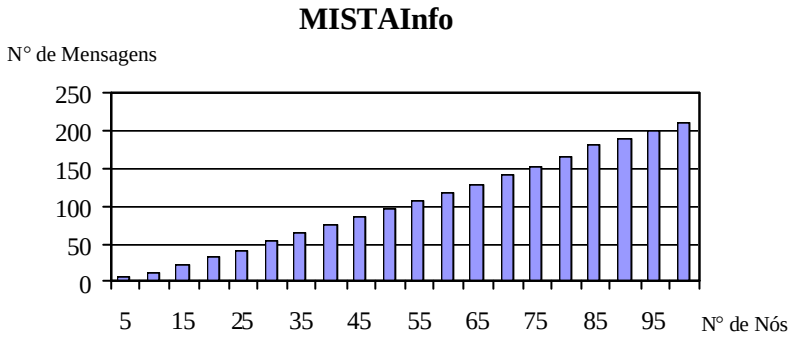


Figura 5.9 - Números médios de mensagens MISTAInfo para a falha e reparação de um enlace

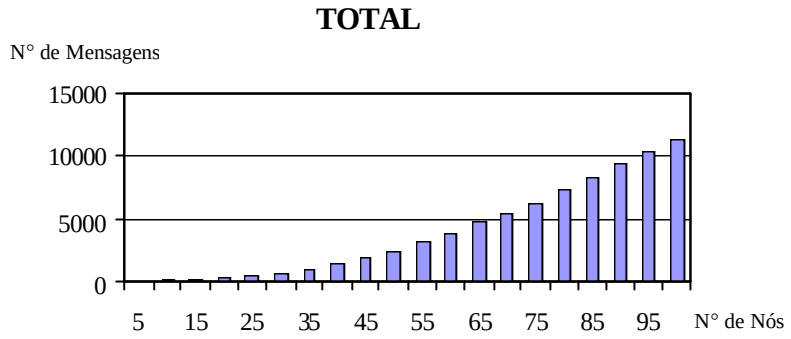


Figura 5.10 - Números médios totais de mensagens para a falha e reparação de um enlace

Observando os gráficos para os números de mensagens geradas, percebe-se que as mensagens dos tipos MESMAInfo e ANTIGAInfo crescem de forma exponencial em função do número de nós da rede. As mensagens dos tipos NOVAInfo e MISTAInfo crescem de forma linear, já que elas são as mensagens estritamente necessárias e suficientes para que cada nó da rede obtenha diagnóstico correto para um evento de falha ou reparação. Como veremos a seguir, as mensagens do tipo MESMAInfo e ANTIGAInfo, podem crescer como quadrado do número de nós.

5.4 Desempenho e Comparações

Faremos a seguir uma exposição comparativa entre o nosso algoritmo e outros similares, sobre os melhores e piores casos para *número de testes por etapa, tráfego de mensagens* e *latência de diagnóstico.* Vale observar que, ao contrário do nosso, alguns dos algoritmos comparados não trabalham em redes de topologia geral, ou ainda não tratam a possibilidade de falhas nos enlaces de comunicação. Em todas as tabelas comparativas, denominaremos o nosso algoritmo por *NetInspector*, o mesmo nome que foi dado à sua implementação prática.

5.4.1 Número de Testes por Etapa

Uma etapa de testes é definida como o período de tempo em que todos os nós participantes do sistema de diagnóstico tenham realizado todos os testes a que são responsáveis.

Numa rede com *N* nós e *L* enlaces, o número mínimo de testes necessários para um procedimento de diagnóstico que considere falhas apenas nos nós é igual a *N* testes por etapa. Alternativamente, o número mínimo de testes necessários para um procedimento de diagnóstico que considere falhas nos nós e também nos enlaces é igual a *L* testes por etapa. Nestas condições, uma topologia de testes ótima é aquela em que cada nó ou enlace é testado uma única vez por etapa.

O emprego de uma topologia de testes ótima, entretanto, pode ocasionar que algumas configurações de falha não sejam detectadas pelo procedimento de diagnóstico. A Figura 5.11 mostra uma situação que ilustra este fato. Nesta figura, cada nó é testado uma única vez (como mostrado pelas arestas orientadas). Se os nós *1* e *2* falharem simultaneamente, *3* e *4* não perceberão que eles estão falhos, e o procedimento de diagnóstico não será correto diante desta circunstância. O mesmo é também verdade se os nós *1*, *2* e *3* falharem simultaneamente, ou seja, o nó *4* não será capaz de diagnosticar corretamente estas falhas. Estas configurações de falhas são denominadas em [RAN 95] por *jellyfish fault node configurations* e podem envolver de um a um número arbitrário de nós.

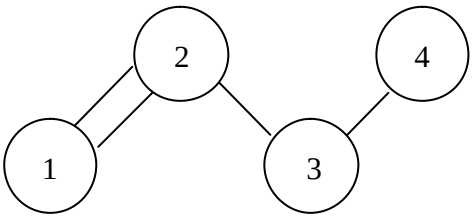


Figura 5.11 - Topologia de testes ótima

Diante destes potenciais problemas, o uso de uma topologia de testes ótima não é algo razoável em um sistema de diagnóstico de falhas em redes de computadores, visto que é inaceitável a possibilidade de que uma porção inteira da rede se torne falha, sem que isto seja detectado pelo sistema de diagnóstico.

Em [DUA 98b] é apresentado um mecanismo para testes denominado de *two-way* que é capaz de manter uma topologia de testes ótima (cada enlace é testado uma única vez por etapa), entretanto, garante que uma configuração de falhas do tipo *jellyfish* seja sempre detectada. Este mecanismo mantém uma topologia de testes mínima (ótima) fazendo com que cada enlace *x-y* seja testado por apenas um dos nós de suas extremidades, digamos *x*. O nó *y*, entretanto, é capaz de monitorar a atividade (freqüência) com que *x* realiza seus testes naquele enlace. Se um tempo máximo aceitável (*threshold*) for estabelecido para o intervalo entre dois testes consecutivos realizados por *x* no enlace *x-y*, o nó *y* poderá perceber que *x* ou o enlace *x-y* estão falhos, caso este tempo máximo aceitável para a espera de um teste seja excedido. Desta forma, *y* sempre notificará uma possível configuração de falha *jellyfish* envolvendo qualquer um de seus vizinhos.

O mecanismo *two-way*, entretanto, gera a propagação de uma mensagem adicional a cada teste realizado. Quando um nó *x* realiza um teste com sucesso no enlace *x-y*, ele envia para *y* uma mensagem adicional contendo o tempo no qual este teste foi realizado. Este tempo contido na mensagem será usado por *y* para monitorar a atividade de testes realizados por *x* no enlace *x-y*.

Em nosso algoritmo, adotamos a topologia de testes bidirecional por enlace, na qual cada enlace é testado pelos dois nós extremidades que ele conecta. Esta estratégia gera apenas uma mensagem a mais do que a *two-way*, para cada etapa de testes em um enlace. Como as mensagens de testes são usualmente pequenas (em nossa implementação prática têm apenas dois *bytes*), acreditamos que a simplicidade no emprego desta estratégia é mais compensadora do que a estratégia *two-way*. Além disso, um leve ganho em latência de diagnóstico é obtido com relação à estratégia *two-way*, visto que um nó não necessita esperar por um tempo usualmente longo entre testes (*threshold*) para perceber que um vizinho, ou o enlace que o conecta a ele, está falho, já que os dois testes são assíncronos.

Desta forma, o número de testes em nosso algoritmo é sempre igual a duas vezes o número de enlaces (*2L*). A Tabela 5.1 mostra os resultados comparativos *em números de testes* do nosso algoritmo com outros similares, mais uma vez lembrando que alguns não tratam falhas nos enlaces (*ADSD*, *Hi-ADSD* e *RDZ*), ou ainda não operam em redes de topologia geral (*ADSD* e *Hi-ADSD*).

O emprego desta topologia de testes, apesar de gerar mais mensagens do que a topologia ótima, é, sem dúvida, mais vantajoso do que a possibilidade de haver circunstâncias de falhas não detectadas na rede (*jellyfish*) ou ainda que um nó seja incorretamente considerado falho devido a uma falha em um de seus enlaces, mesmo que ainda seja possível alcançá-lo por um caminho alternativo.

NÚMERO DE TESTES POR ETAPA			
ALGORITMO	**MELHOR CASO**	**PIOR CASO**	**FONTE**
SELF	$N(t+1)^*$	$N(t+1)^*$	[BIA 92]
Adaptive DSD	*N*	*N*	[BIA 92]
Hi-ADSD	*N*	*N*	[DUA 98]
*Adapt***	*L*	*2L*	[STA 92]
RDZ	*N*	*N*	[RAN 95]
DNMN	*L*	*L*	[DUA 98b]
NetInspector	***2L***	***2L***	—

* Para uma rede em que no máximo *t* unidades podem falhar.
** Versão que trata falhas em enlaces.

Tabela 5.1 - Comparativo em número de testes por etapa

5.4.2 Tráfego de Mensagens

Para analisar o tráfego de mensagens do nosso algoritmo, consideraremos os melhores e piores casos para o tráfego de mensagens gerado diante da falha de um único nó ou de um único enlace, considerando uma rede com *N* nós.

O melhor caso para o tráfego de mensagens devido a uma falha em nó no nosso algoritmo é aquele em que um único vizinho detecte a falha e inicie a propagação de mensagens notificando o evento. Ora, o número mínimo de mensagens geradas neste caso é igual a ($N - 2$), visto que nenhuma mensagem será propagada para o nó falho, nem para aquele que detectou a falha. Na verdade, o nó que detectou a falha iniciará a propagação das mensagens, e pelo menos ($N - 2$) mensagens serão necessárias para notificar o evento a todos os demais nós normais remanescentes.

O pior caso para o tráfego de mensagens diante da falha de um nó ocorre em redes completamente conectadas. Neste caso, se um nó sofre uma falha, e todos os demais nós normais remanescentes ($N - 1$) detectam-na simultaneamente, serão propagadas um total de ($N - 1$)($N - 2$) mensagens, já que, obviamente, um nó não propaga uma mensagem para si próprio. Desta forma, dependendo da topologia de interligação da rede, os piores casos para o tráfego de mensagens são $\boldsymbol{O}(N^2)$.

Para a falha de um enlace, o melhor caso ocorre em redes de conexão mínima, quando há a separação dos dois nós das extremidades que ele conecta em dois novos componentes normais conexos. Neste caso, o número mínimo de mensagens é igual a ($N - 2$) mensagens. A Figura 5.12 ilustra esta situação. Na figura, o valor mínimo para ($N_x + N_y$) é igual a ($N - 2$).

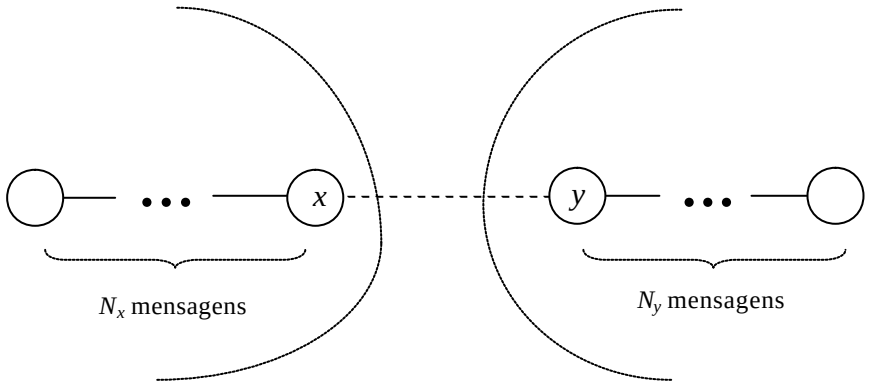


Figura 5.12 - Falha de enlace que gera dois novos componentes normais conexos

Se a falha no enlace não ocasiona que os dois nós das extremidades que ele conecta fiquem separados em dois componentes normais conexos distintos, então o *número mínimo de mensagens necessárias e suficientes* para que todos os nós obtenham um diagnóstico correto para este fato é diretamente proporcional ao tamanho do menor *caminho normal* entre estes dois nós. Em outras palavras, quanto maior a distância entre estes dois nós, depois da falha, maior o número de *mensagens necessárias* que devem ser propagadas através da rede.

A Figura 5.13 mostra o caso em que o maior número destas mensagens será propagado, desde que os nós estão diametralmente opostos, isto é, o menor caminho entre eles percorre todos os demais nós da rede.

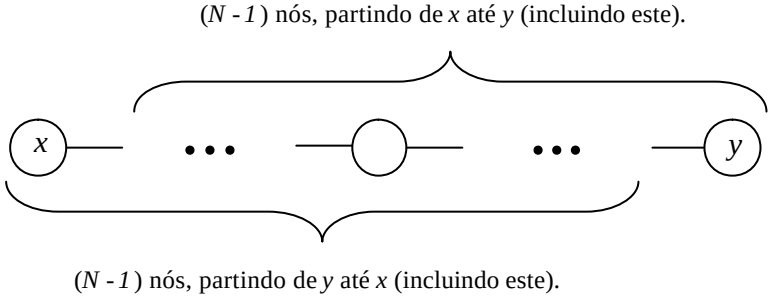


Figura 5.13 - Maior distância entre os nós *x* e *y* após a falha no enlace *x*-*y*

Observando a Figura 5.13, e recordando o que foi apresentado no *Capítulo 4*, *x* e *y* irão eventualmente receber a mensagem originada por um *notificando a falha* do outro. Ao receber a mensagem, eles originarão uma outra, dita *retificadora*, reafirmando os seus estados corretos. Deste modo, quatro mensagens distintas serão geradas, e terão que percorrer, cada uma delas, (*N* - *1*) nós, que é o tamanho do menor caminho entre *x* e *y* (incluindo um dos extremos). Assim, o menor número de mensagens necessárias (e suficientes) para um diagnóstico correto nestas circunstâncias é igual a *4*(*N* - *1*) mensagens.

A Figura 5.14 mostra o caso em que o menor número de mensagens necessárias será gerado, desde que *x* e *y* estão o mais próximo possível um do outro após a falha do enlace que os conectava.

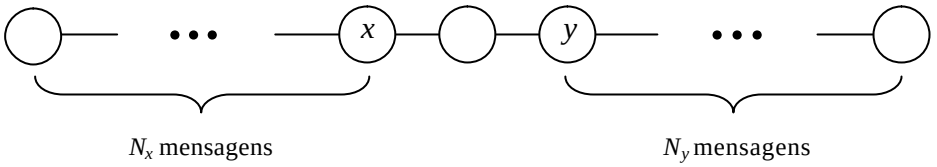


Figura 5.14 - Menor distância entre os nós *x* e *y* após a falha no enlace *x*-*y*

Observando a Figura 5.14, e também recordando o que foi apresentado no *Capítulo 4*, *x* e *y* receberão mais cedo as mensagens de um notificando a falha do outro, e, deste modo, haverá uma redução no número de mensagens necessárias e suficientes para um diagnóstico correto em todos os nós. Isto é fácil de se verificar observando que *x* e *y* não propagarão adiante, para N_x e N_y nós respectivamente, as mensagens originais que notificavam suas próprias falhas. Assim, um total de $4(N - 1) - (N_x + N_y)$ mensagens serão necessárias (e suficientes) para um diagnóstico correto desta falha de enlace, em todos os nós. Como $(N_x + N_y)$ é igual ao número total de nós menos *3* (*x*, *y* e o nó entre eles), o número mínimo de mensagens necessárias e suficientes para a falha em um enlace nestas circunstâncias é igual a $4(N - 1) - (N - 3) = (3N - 1)$.

O nosso algoritmo, entretanto, devido a sua estratégia de propagação de mensagens em paralelo, gera quase sempre um número maior do que o mínimo suficiente de mensagens para um diagnóstico correto de falha em enlace, separando ou não o sistema original em dois novos componentes normais conexos. À exemplo do que pode ocorrer com as falhas de nós, uma falha em enlace pode gerar um número de mensagens em $\boldsymbol{O}(N^2)$, no pior caso. De um modo geral, as falhas que não separam o sistema original em mais de um componente normal conexo são sempre as que geram o maior tráfego de mensagens na rede.

A Tabela 5.2 mostra os resultados comparativos em *tráfego de mensagens*.

TRÁFEGO DE MENSAGENS					
	FALHA EM UM NÓ		FALHA EM UM ENLACE		
ALGORITMO	MELHOR CASO	PIOR CASO	MELHOR CASO	PIOR CASO	FONTE
SELF	$O(Nt^2)^*$	$O(N^2t^2)^*$	__	__	[BIA 92]
Adaptive DSD	*1*	*N*	__	__	[BIA 92]
Hi-ADSD	*N*	*N*	__	__	[DUA 98]
Adapt	*3(2N-3)*	$O(N^2)$	*3(2N-3)*	$O(N^2)$	[STA 92]
RDZ	(*N-2*)	$O(N^2)$	__	__	[RAN 95]
DNMN	(*N-2*)	$O(N^2)$	(*N-2*)	$O(N^2)$	[DUA 98b]
NetInspector	**(*N-2*)**	$\mathbf{O(N^2)}$	**(*N-2*)**	$\mathbf{O(N^2)}$	__

* Para uma rede em que no máximo *t* unidades podem falhar.

Tabela 5.2 - Comparativo em tráfego de mensagens

5.4.3 Latência de Diagnóstico

A *latência de diagnóstico* é o tempo decorrido entre a detecção de um novo evento de falha ou reparação até o momento em que todos os nós do componente normal conexo, ao qual o evento pertença, tenham adquirido diagnóstico correto para ele. Usando a estratégia de propagação de mensagens em paralelo, uma redução significativa na latência de diagnóstico é conseguida.

Usando a estratégia de propagação de mensagens em paralelo, a latência torna-se ótima e diretamente proporcional ao diâmetro da rede, isto é, à maior distância existente entre dois nós. Numa rede com *N* nós, a distância entre quaisquer dois nós pode variar de *1* (se eles forem vizinhos) a, no máximo, (*N - 1*), se eles forem diametralmente opostos.

Desta forma, os melhores casos para a latência de diagnóstico ocorrem em redes completamente conectadas, visto que a maior distância entre quaisquer dois nós é sempre igual a *1*. Por outro lado, os piores casos podem ocorrer, potencialmente, em redes de conexão mínima, visto que a distância entre dois nós pode chegar a (*N - 1*).

No nosso algoritmo, os piores casos potenciais para latência de diagnóstico ocorrem quando a falha em um enlace não separa os dois nós que ele conecta em componentes normais conexos distintos, pois, neste caso, serão propagadas quatro mensagens distintas.

Usando o mesmo raciocínio exposto sobre o tráfego de mensagens diante de falhas em nós e enlaces, é fácil verificar os valores para latência de diagnóstico em cada caso.

A Tabela 5.3 mostra os resultados comparativos em *latência de diagnóstico.*

LATÊNCIA DE DIAGNÓSTICO					
	FALHA EM NÓ		**FALHA EM ENLACE**		
ALGORITMO	**MELHOR CASO**	**PIOR CASO**	**MELHOR CASO**	**PIOR CASO**	**FONTE**
SELF	$(N/(t+1))/T_{test}^{*}$	$(N/(t+1))/T_r^{*}$	—	—	[BIA 92]
Adaptive DSD	NT_{test}	NT_r	—	—	[BIA 92]
Hi-ADSD	$\lceil Log_2 N \rceil T_r$	$\lceil (Log_2 N)^2 \rceil T_r$	—	—	[DUA 98]
Adapt	NT_r	$O(N^2 T_r)$	NT_r	$O(N^2 T_r)$	[STA 92]
RDZ	T_m	$T_m(N-2)$	—	—	[RAN 95]
DNMN	T_m	$T_m(N-2)$	T_m	$T_m(N-1)$	[DUA 98b]
NetInspector	$\boldsymbol{T_m}$	$\boldsymbol{T_m(N-2)}$	$\boldsymbol{T_m(N-2)}$	$\boldsymbol{4T_m(N-1)}$	—

* Para uma rede em que no máximo *t* unidades podem falhar.

T_{test} = Tempo médio necessário para um nó realizar um teste em um vizinho.
T_r = Tempo médio decorrido entre duas *etapas de testes* consecutivas. (*testing rounds*)
T_m = Tempo médio necessário para enviar uma mensagem a um vizinho.

Tabela 5.3 - Comparativo em latência de diagnóstico

CAPÍTULO 6

Implementação

6.1 Introdução

Neste capítulo, apresentamos uma implementação prática do nosso algoritmo de diagnóstico de falhas em redes, usando a linguagem *Java.*

Na seção 6.2, faremos uma breve apresentação da linguagem, exaltando alguns dos principais pontos que motivaram o seu uso na implementação.

Na seção 6.3, apresentaremos o programa *NetInspector*, um programa para diagnóstico de falhas em redes de topologia geral. Apresentaremos uma visão geral do funcionamento do programa, alguns detalhes de sua implementação e ainda os arquivos de configuração necessários à sua execução.

6.2 Linguagem *Java*

A linguagem *Java*™ [GOS 95] tem surgido como uma excelente escolha para o desenvolvimento de aplicações distribuídas, multiplataformas e orientadas a objetos.

Apresentando uma sintaxe similar à de linguagens tradicionais para este tipo de desenvolvimento, como *C* e *C++*, *Java* conquistou rapidamente inúmeros adeptos, sendo, atualmente, uma linguagem já madura e bastante promissora.

Java é uma linguagem robusta e portável, cujo código executável é compilado para uma plataforma de *arquitetura neutra*, denominada *máquina virtual*. Uma vez compilado o código fonte, o mesmo código objeto (*bytecodes*) pode ser executado em qualquer ambiente onde haja implementada uma plataforma de máquina virtual.

Atualmente, encontram-se implementações destas plataformas para qualquer sistema operacional popular, como, por exemplo, *Windows 95/98/NT*, *MacOS, System 7*, *Linux*, *SunOS, Solaris, IBMAix* e *HPUX*. Desta forma, a heterogeneidade dos equipamentos (estações) que compõem a rede torna-se praticamente irrelevante.

Além da portabilidade, a linguagem *Java* é totalmente orientada a objetos, facilitando o desenvolvimento das aplicações e permitindo reutilização de código. O seu ambiente de desenvolvimento agrega diversas classes para criação de *threads*, interfaces gráficas e um conjunto completo de métodos para a comunicação entre processos através da rede, usando as interfaces *Berkeley Sockets.*

Uma vantagem adicional é a possibilidade de usar *Web browsers* para executar programas feitos em *Java*, algo que tem se tornando bastante usual como interface gráfica de sistemas de gerenciamento de falhas.

Por todos estes fatores, elegemos a linguagem *Java* como o ambiente mais propício para o desenvolvimento de nossa implementação.

6.3 *NetInspector* - Diagnóstico de Falhas em Redes de Topologia Geral

O programa *NetInspector* é capaz de detectar e localizar falhas permanentes do tipo *fail-stop* em um sistema composto por *N* estações de trabalho (*workstations*) ligadas através de uma rede de topologia geral, utilizando o protocolo de comunicação TCP/IP. Cada estação deve possuir um ambiente *Java Runtime* (*máquina virtual*) para poder executar o programa.

Inicialmente, é necessário compor um arquivo com informações sobre todas as *N* estações (*hosts*) que participarão do sistema de diagnóstico. Cada entrada deste arquivo corresponde a uma estação participante do sistema, e contém um identificador para a estação e o seu endereço IP. O identificador é um número inteiro que varia de zero (*0*) ao número de estações menos uma (*N-1*). Este arquivo, denominado SYSHOSTS.INF, deve ser sempre único e estar disponível para todas as estações participantes do sistema de diagnóstico.

Cada estação, por sua vez, deve manter um outro arquivo localmente, que é diferente para cada uma delas, denominado NODECFG.INF. Este arquivo diz a cada estação qual o seu próprio identificador e quais os identificadores de suas vizinhas.

É importante verificar que a topologia (lógica) de interligação das estações no sistema é determinada direta e unicamente pelos arquivos NODECFG.INF existentes em cada estação. Estes arquivos devem ser consistentes, no sentido de que se a estação *x* tem no seu arquivo de configuração NODECFG.INF a informação de que é vizinha da estação *y*, então *y* também deve ter uma informação, em seu próprio arquivo, configurando que é vizinha de *x*.

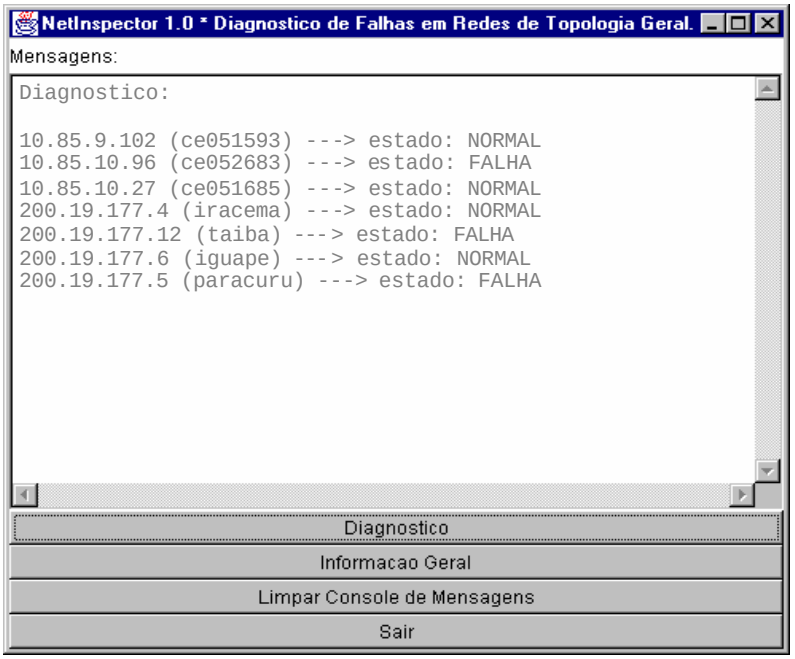


Figura 6.1 - Interface gráfica do programa *NetInspector* 1.0

A partir destes arquivos, o impacto de acrescentar ou remover uma estação *x* ao sistema de diagnóstico é mínimo, bastando acrescentar ou remover uma entrada correspondente a *x* no arquivo SYSHOSTS.INF e alterar os arquivos NODECFG.INF das estações vizinhas de *x*, acrescentando-a (ou removendo-a) como vizinha em cada uma delas. Em outras palavras, o impacto de acrescentar ou remover uma estação ao sistema

de diagnóstico é localizado e diretamente proporcional ao número de suas estações vizinhas.

A Figura 6.1 mostra a interface gráfica do programa *NetInspector* 1.0. A interface é simples e composta basicamente de uma *console de mensagens*, e alguns *botões de ação*.

Na *console de mensagens* são apresentadas mensagens de texto que podem ser motivadas pelo pressionamento de algum dos botões de ação. Os botões de ação são os seguintes:

- *Diagnóstico* - Mostra a *situação de falha* da rede. O pressionamento deste botão causa a exibição na console de mensagens de uma lista com cada uma das *N* estações participantes do sistema de diagnóstico (os seus endereços IP) associada à palavra FALHA ou NORM AL.
- *Informação Geral* - Mostra informações sobre o sistema e uma lista com dados sobre as estações vizinhas. O pressionamento deste botão causa a exibição na console de mensagens de todas as informações existentes nos arquivos SYSHOSTS.INF e NODECFG.INF.
- *Limpar Console de Mensagens* - Remove todas as mensagens da console.
- *Sair* - Encerra o funcionamento do programa.

O programa *NetInspector* pode ser executado em *background* e ficar monitorando continuamente a situação de falha da rede, que pode ser consultada a qualquer momento, pressionando o botão *Diagnóstico*.

As estações executando o programa trocam pacotes com informações de diagnóstico através da rede utilizando as interfaces *Berkeley Sockets*. [STE 90]

Foram utilizados *threads* na implementação para executar as tarefas que podem ser feitas de uma forma independente e *quase paralela*. A Figura 6.2 mostra os cinco tipos de *threads* de execução que compõem o programa *NetInspector*.

As tarefas *2,3* e *4* são executadas de forma que haja tantas delas quantas forem as vizinhas da estação executado o programa (N_v). A tarefa *5*, embora seja executada em

paralelo às demais, atende às solicitações das vizinhas de forma sequencial, garantindo a consistência da atualização dos dados de diagnóstico mantidos localmente.

As tarefas de *realizar e responder a testes* e *enviar e receber mensagens* foram modeladas como *classes* independentes e com interfaces genéricas - o que facilita a compreensão da concepção e modularidade do programa. Uma outra conseqüência deste fato, é que estas classes podem ser adaptadas facilmente para trabalhar com outras tecnologias de rede, que não sejam baseadas em TCP/IP, sem a necessidade de alterar a funcionalidade do núcleo principal do programa.

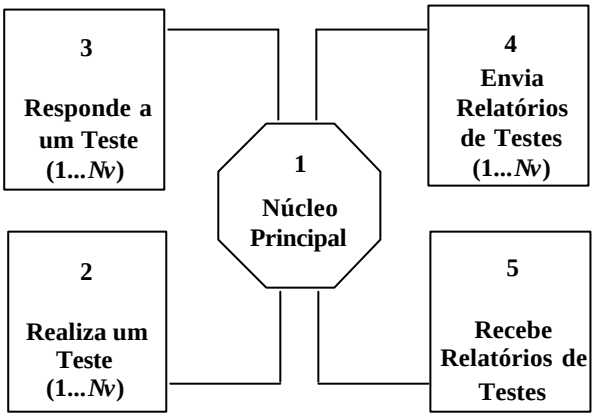


Figura 6.2 - Tipos de *threads* que compõem a execução do programa *NetInspector*

O sistema de diagnóstico torna a topologia de interligação real da rede transparente para as estações participantes executando o programa *NetInspector*. Em outras palavras, o sistema de diagnóstico funciona como uma camada na qual a rede de comunicação física real é mapeada numa rede ponto a ponto lógica. A rede física pode ser composta de enlaces ponto a ponto, canais baseados em difusão (*broadcast*) ou uma combinação arbitrárias destes. A rede lógica, construída pelo sistema de diagnóstico, faz com que a topologia de interligação comporte-se sempre com uma rede ponto a ponto, apenas, naturalmente, para efeito de testes e troca de mensagens de diagnóstico entre as estações, ou seja, apenas para efeito do próprio funcionamento do sistema de diagnóstico.

Inicialmente, é preciso determinar a topologia lógica de interligação da rede, definindo que estações serão vizinhas dentro do sistema de diagnóstico. Como visto no *Capítulo 4*, se duas estações estão conectadas através de um enlace ponto a ponto, elas serão necessariamente *vizinhas físicas*. Se duas estações estão conectadas através de um canal *brodcast* (ex. *Ethernet*) elas *podem ou não* ser convencionadas *vizinhas lógicas* dentro do sistema de diagnóstico, através dos seus arquivos NODECFG.INF, como visto anteriormente.

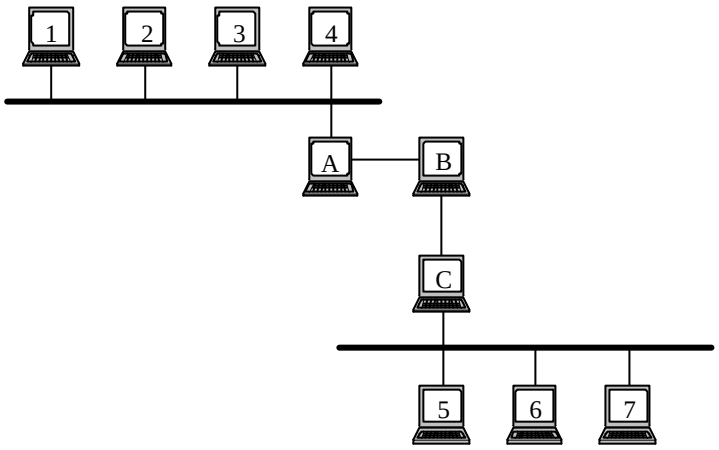


Figura 6.3 - Redes *broadcast* conectadas através de uma rede ponto a ponto

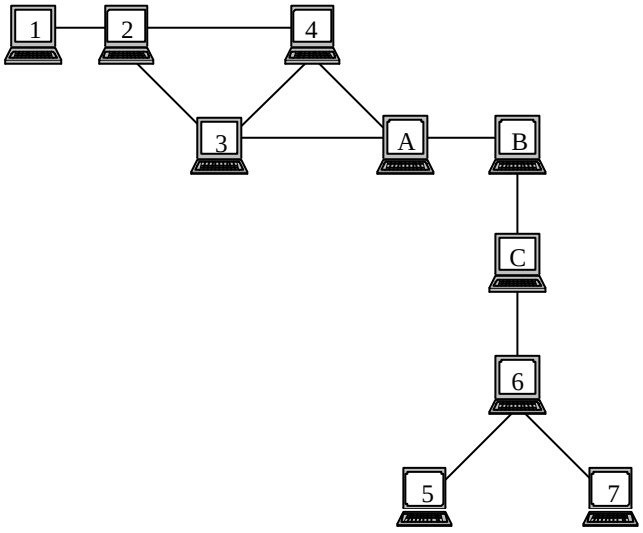


Figura 6.4 - Possível rede de interligação lógica para o sistema de diagnóstico

A Figura 6.3 mostra uma rede de comunicação física real, na qual as estações com identificadores de 1 a 7 estão conectadas a redes *broadcast*. As estações com identificadores A e C estão conectadas tanto a redes *broadcast* quanto a uma rede ponto a ponto. A Estação B está conectada apenas à rede ponto a ponto.

Através dos arquivos de configuração do sistema de diagnóstico, uma possível configuração lógica para a rede é mostrada pela Figura 6.4. É importante observar que nas redes *broadcast* é possível se fazer qualquer tipo de configuração de vizinhança (lógica) entre as estações que a compõem, o que não ocorre nas redes ponto a ponto.

Uma falha *fail-stop* de uma estação pode ser simulada pelo programa *NetInspector*, fazendo com que todos os *threads* que realizam e respondem a testes, e ainda todos os *threads* que recebem e enviam mensagens de diagnóstico, sejam desativados na estação correspondente, como ilustra a Figura 6.5. Por outro lado, uma falha *fail-stop* de um enlace pode ser simulada pelo programa fazendo com que apenas os *threads* para responder a testes e receber mensagens relativos às duas estações vizinhas que ele conecta (física ou logicamente) sejam desativados, como ilustra a Figura 6.6.

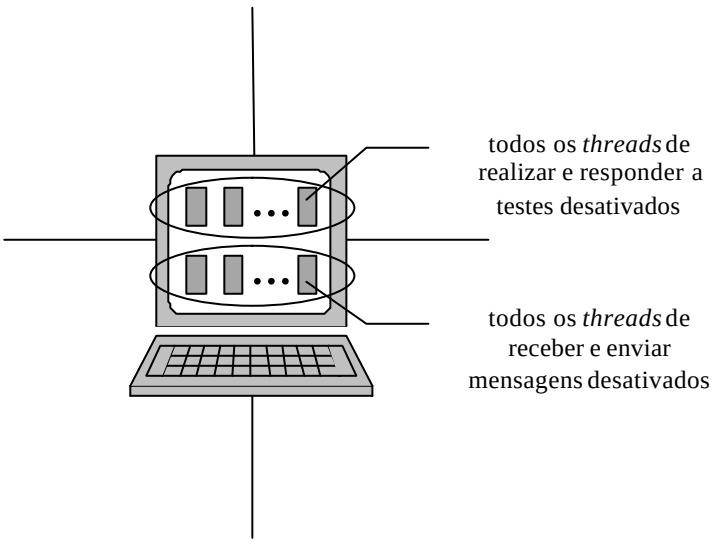


Figura 6.5 - Simulação de falha *fail-stop* de uma estação

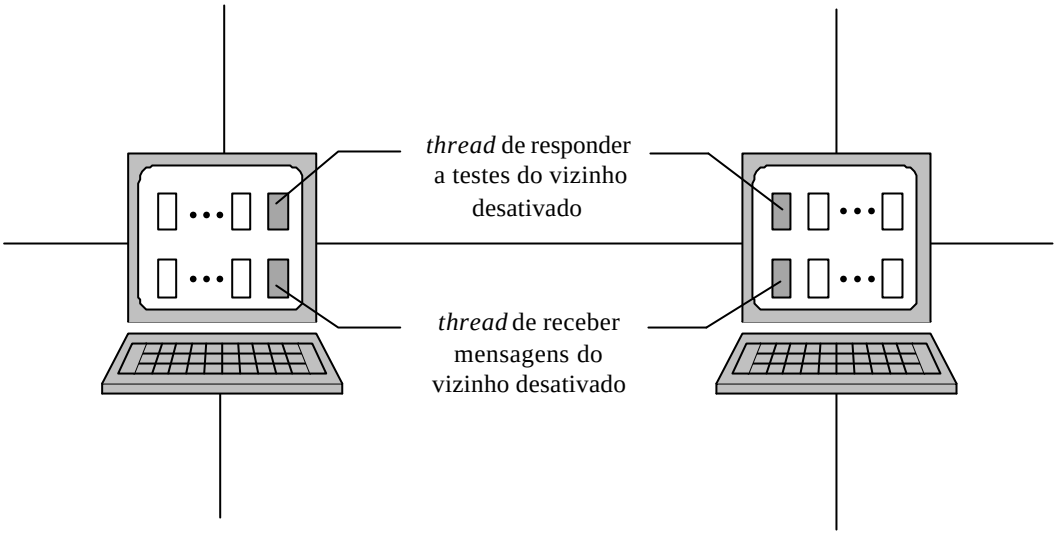


Figura 6.6 - Simulação de falha *fail-stop* de um enlace

O *Apêndice B* contém o código fonte do programa *NetInspector* e um exemplo de sistema de diagnóstico real com os respectivos arquivos SYSHOSTS.INF e NODECFG.INF. No código fonte estão comentados em detalhes as estruturas de dados e as classes do programa.

CAPÍTULO 7

Conclusões

7.1 Relevância do Trabalho

A área de *Diagnóstico Automático de Sistema* tem sido alvo de diversos trabalhos de pesquisa nos últimos anos. Dentre estes trabalhos, estamos particularmente interessados nos que propõem *algoritmos para diagnóstico distribuído de falhas em redes de computadores.*

De acordo com a nossa revisão da literatura técnica, há algoritmos que só trabalham com falhas nos processadores. [RAN 95] [BIA 92] [DUA 98] Outros tratam a possibilidade de falhas nos enlaces de comunicação, mas exigem que cada nó conheça toda a topologia de interligação do sistema [DUA 98b], ou empregam o tráfego de grandes quantidades de mensagens diante de qualquer falha ou reparação. [STA 92]

Alguns algoritmos só funcionam em redes completamente conectadas. [BIA 92] [DUA 98] Outros trabalham em redes de topologia geral mas funcionam *off-line*, isto é, não operam corretamente no caso de falhas ou reparações ocorrerem enquanto estão sendo executados. [BAG 91]

Nesta dissertação, apresentamos um algoritmo para *diagnóstico distribuído* de falhas em *processadores* conectados através de uma *rede de topologia geral*, que funciona *on-line*, permitindo falhas e reparações tanto nos *nós* quanto nos *enlaces*. O algoritmo *é correto em caso de falhas nos enlaces de comunicação*, e *não exige que cada nó conheça toda a topologia de interligação do sistema.* O tráfego de mensagens gerado pela sua execução, em ordem de grandeza no pior caso, é equivalente ao de algoritmos similares, como em [RAN 95] e [DUA 98b]. Acreditamos que este conjunto de características do algoritmo é a principal contribuição deste trabalho.

O algoritmo proposto, no entanto, emprega uma topologia de testes máxima (cada nó testa todos os seus vizinhos), e apresenta ainda um acréscimo no número de mensagens redundantes, propagadas para notificar eventos de falhas e reparações.

Apresentamos a prova formal do corretismo do algoritmo, e alguns resultados obtidos através de simulação de sua execução em redes de conexão mínima, completamente conectada e de topologia geral. Tabelas comparativas entre seus parâmetros de desempenho e os de outros algoritmo similares foram também apresentadas.

Finalmente, apresentamos ainda um código implementado do algoritmo e um exemplo de sistema de diagnóstico real, o que também é uma contribuição relevante.

7.2 Trabalhos Futuros

7.2.1 Implementação baseada em SNMP

Um prosseguimento deste trabalho seria implementá-lo usando informações de uma MIB SNMP.

Alguns autores descrevem a implementação de algoritmos de diagnóstico automático de sistema baseadas no protocolo padronizado de gerência de redes *Simple Network Management Protocol* (SNMP). [DUA 98,98b] Estas implementações são feitas codificando tabelas e outras estruturas de dados utilizadas pelo algoritmo em ASN.1, e integrando-as a uma MIB (*Management Informantion Base*) SNMP.

Os algoritmos seriam executados no topo do protocolo SNMP, utilizando seus serviços para comunicação entre dois nós participantes do sistema de diagnóstico. O uso de *traps*, por exemplo, é proposto com o objetivo de diminuir a latência de diagnóstico, já que eles ocorrem de forma assíncrona, sempre que um novo evento deve ser notificado ao monitor (gerente) SNMP. Isto permite que novos eventos de falha ou reparação sejam detectados em um nó antes mesmo que um próximo teste convencional seja realizado sobre ele.

O uso de *traps* e outros serviços do protocolo SNMP em nosso algoritmo poderia nos dar a opção de diagnosticar outros tipos de falhas, que não fossem simplesmente a estação participante do sistema de diagnóstico cessar sua operação permanentemente. Por exemplo, poderíamos definir uma falha como sendo um estouro no número máximo de pacotes que estejam trafegando na porta de um roteador. O acesso à MIB SNMP nos permite verificar vários destes parâmetros, e se eles indicam configurações de falha ou situação anômala na rede.

As redes reais não se compõem apenas de *hosts.* Existem diversos outros dispositivos, como roteadores, pontes, impressoras, modems, terminais, etc. Um outro uso importante para o algoritmo seria a possibilidade de diagnosticar falhas não apenas nos *hosts*, que são capazes de realizar processamento e testes, mas também nestes outros dispositivos conectados à rede. Estes dispositivos não são capazes de realizar testes, mas podem ser testados como objetos gerenciados através de parâmetros na MIB SNMP. [DUA 98]

7.2.2 Integração a um Sistema de Gerência de Redes (SGR)

Sistemas de gerenciamento de redes baseados em protocolos padronizados (SGRs) oferecem mecanismos práticos para o monitoramento e controle de redes de computadores, objetivando a melhoria dos seus serviços, aumento de disponibilidade, e redução dos custos de operação, tendo ainda sido alvos de inúmeros esforços, feitos por órgãos de normalização internacionais, com o intuito de propor padrões efetivos que garantam a interoperabilidade entre soluções por ventura apresentadas por fabricantes diferentes. Tais sistemas são os candidatos naturais para plataformas que permitam a detecção de situações anormais (falhas) que comprometam o funcionamento de redes de computadores.

Estes sistemas de gerenciamento são usualmente concebidos segundo o modelo *gerente/agente*, no qual um processo específico denominado *gerente* é responsável por coletar e enviar informações de gerenciamento a outros processos denominados *agentes*, num modelo funcional que aproxima-se do paradigma empregado na arquitetura *cliente/servidor*. Os gerentes são responsáveis pelas ações de gerenciamento

propriamente ditas, enquanto que os agentes são responsáveis por coletar dados operacionais e detectar eventos excepcionais em objetos gerenciados que são abstrações de recursos reais e estão devidamente representados em uma base de dados de informações de gerenciamento denominada MIB (*Management Information Base*).

Embora este modelo proporcione um *framework* prático no qual as atividades de gerenciamento possam ser executadas de forma modular, ele apresenta sérios problemas de *confiabilidade,* dado que uma *falha na estação onde é executado o gerente* causa a parada das ações de gerenciamento na porção inteira da rede monitorada por ele.

Em particular, algumas situações de falha podem causar a parada completa deste tipo de sistema, o que, naturalmente, é algo extremamente indesejável quando a disponibilidade da rede é essencial.

Um trabalho futuro a ser realizado é a integração de nosso algoritmo de diagnóstico a um SGR, já que o algoritmo é tolerante à existência de múltiplas falhas, o que não ocorre no modelo convencional gerente/agente empregado pelo SGR.

Em [DUA 98b] apresenta-se a integração de um algoritmo de diagnóstico automático de sistema a um sistema de gerência de redes, e alguns resultados experimentais obtidos em uma implementação prática para uma rede local (LAN).

Neste trabalho são analisados o impacto desta abordagem de integração no tráfego da rede e na latência de diagnóstico. São discutidos ainda os problemas em se usar os serviços do protocolo de gerenciamento SNMP como base para a comunicação entre os nós do sistema, já que este protocolo utiliza um mecanismo de comunicação não confiável, baseado no protocolo UDP (*User Datagram Protocol*).

REFERÊNCIAS

[AND 81] ANDERSON, T.; LEE, P. A., *Fault Tolerance - Principles and Practice*, Prentice Hall, 1981.

[BAG 91] BAGCHI, A.; HAKIMI, S. L., "An Optimal Algorithm for Distributed System-Level Diagnosis," in *Proc. 21 th Fault-Tolerant Computing Symp.*, June, 1991.

[BAR 76] BARSI, F.; GRANDONI, F.; MAESTRINI, P., "A Theory of Diagnosability of Digital Systems," *IEEE Transactions on Computers*, vol. C-25, no. 6, pp. 585-593, 1976.

[BIA 90] BIANCHINI, R. P.; GOODWIN, K.; NYDICK, D. S., "Practical Application and Implementation of Distributed System-Level Diagnosis Theory," in *Proc. Twentieth Int. Symp. Fault-Tolerant Comput.*, IEEE, June 1990, pp. 332-339.

[BIA 92] BIANCHINI, R. P.; BUSKENS, R., "Implementation of On-Line Distributed System-Level Diagnosis Theory," *IEEE Transactions on Computers*. vol 41. pp. 616-626, 1992.

[BIL 82] BILLINGTON, R; ALLAN, R. N., *Reliability Evaluation of Engineering Systems: Concepts and Techniques*, Plenum Press, N.Y., 1982.

[BON 76] BONDY, J. A.; MURTY, U. S. R.; *Graph Theory and Applications*, Elsevier North Holland, Inc., New York, N. Y., 1976.

[DAH 84] DAHBURA, A. T.; MASSON, G. M., "An $O(n^{2.5})$ Fault Identification Algorithm for Diagnosable Systems," *IEEE Transactions on Computers*, vol. c-33, no. 6. pp. 486-492, June 1984.

[DUA 95] DUARTE, E. P. Jr.; NANYA, T., "Multi-Cluster Adaptive Distributed System-Level Diagnosis Algorithms," *IEICE Technical Report FTS 95-73*, 1995.

[DUA 96] DUARTE, E. P. Jr.; NANYA, T., "An SNMP-based Implementation of the Adaptive DSD Algorithm for LAN Fault-Management," in *Proc. IEEE/IFIP NOMS'96*, pp. 530-539, Kyoto, April 1996.

[DUA 98] DUARTE, E. P. Jr.; NANYA, T., "A Hierarchical Adaptative Distributed System-Level Diagnosis Algorithm," IEEE Transactions on Computers, vol 47, no.1, 1998.

[DUA 98b] DUARTE, E. P. Jr.; NANYA, T.; MANSFIELD, G.; NOGUSHI, S., "Non-Broadcast Network Fault-Monitoring Based on System-Level Diagnosis," in *Proc. IEEE/IFIP NOMS'98*, pp. 597-609, 1998.

[GOS 95] GOSLING, J.; MCGILTON, H., *The Java Language Environment - A White Paper*, Sun Microsystems, Oct, 1995.

[HAK 74] HAKIMI, S. L.; AMIN, A. T., "Characterization of Connection Assignment of Diagnosable Systems," *IEEE Transactions on Computers*, vol C-23, Jan, 1974.

[HAK 84] HAKIMI, S. L.; NAKAJIMA, K., "On Adaptive System Diagnosis," *IEEE Transactions on Computers*, vol C-33, pp. 234-240, Mar, 1984.

[HAK 84b] HAKIMI, S. L.; SCHMEICHEL, E. F.; "An Adaptive Algorithm for System-Level Diagnosis," *J. Algorithms.* no. 5. pp. 526-530. 1984.

[HOS 84] HOSSEINI, S. H.; KUHL, J. G.; REDDY, S. M., "A Diagnosis Algorithm for Distributed Computing Systems with Dynamic Failure and Repair," *IEEE Transactions on Computers*, vol. c-33, no. 3. pp. 223-233, March 1984.

[KER 88] KEROLA, T., *SMPL Simulation Subsystem - User Guide*, University of Helsinki, 1988.

[KUH 80] KUHL, J. G.; REDDY, S. M., "Distributed Fault-Tolerance for Large Multiprocessor Systems," in *Proc. 7th Annu. Symp. Comput. Architeture*, IEEE, May 1980, pp. 23-30.

[KUH 81] KUHL, J. G.; REDDY, S. M., "Fault-Diagnosis in Fully Distributed Systems," in *Proc. 11th Int.. Symp. Fault-Tolerant Computing*, pp. 100-105, June 1981.

[MAC 80] MACDOUGALL, M. H., *SMPL - A Simple Portable Simulation Language*, Amdahl Corp, Technical Report April 1, 1980.

[MAC 87] MACDOUGALL, M. H., *Simulating Computer Systems: Techinques and Tools*, Cambridge, Mass.: The MIT Press 1987.

[MAL 80] MALEK, M. “A Comparison Connection Assignment for Diagnosis of Multiprocessor Systems,” in *Proceedings of the Seventh Symposium on Computer Architecture*, pp. 31-35, 1980.

[MAN 96] MANSFIELD, G.; OUCHI, M.; JAYANTHI, K.; KIMURA Y.; OHTA K.; NEMOTO Y., "Techniques for Automated Network Map Generation Using SNMP," in *Proc. of INFOCOM'96*, pp. 473-480, March 1996.

[NAK 81] NAKAJIMA, K., "A New Approach to System Diagnosis," in *Proc. 19th Annu. Allerton Conf. Commun., Contr. and Comput.,* Sept. 1981, pp. 697-706.

[PRE 67] PREPARATA, F. P.; METZE, G.; CHIEN, R. T., "On the Connection Assignment Problem of Diagnosable Systems," *IEEE Trans. Electron. Comput.,* vol. EC-16, pp. 848-854, Dec, 1967.

[RAN 95] RANGARAJAN, S.; DAHBURA, A. T.; ZIEGLER, E. A., "A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies," *IEEE Transactions on Computers,* vol 44, pp. 312-333, 1995.

[SCH 83] SCHLICHTING, R.; SCHNEIDER, F., "Fail-stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Trans. Comput. Syst.* vol. 1, no. 3, pp. 222-238, Aug, 1983.

[SCH 95] SCHWETMAN, H. D., *CSIM17User's Guide,* Mesquite Software, Inc., Austin, Texas, 1995.

[SIE 92] SIEWIOREK, D. P.; SWARZ, R. S., *Reliable Computer Systems: Design and Evaluation,* Second Edition, Digital Press, 1992.

[SOM 87] SOMANI, A. K.; AGARWAL, V. K.; AVIS, D., "A Generalized Theory for System Level Diagnosis," *IEEE Transactions on Computers,* vol. c-36, no. 5. pp. 538-546, May 1987.

[STA 92] STAHL, M.; BUSKENS, R.; BIANCHINI, R., "On-Line Diagnosis on General Topology Networks," in *Proc. Workshop Fault-Tolerant Parallel and Distributed Systems,* July 1992.

[STA 93] STALLINGS W., *SNMP, SNMPv2, and CMIP The Practical Guide to Network-Management Standards,* 1993.

[STE 90] STEVENS, W. R., *Unix Network Programming,* Prentice Hall Software Series, 1990.

[SUL 84] SULLIVAN, G., "A Polynomial Time Algorithm for Fault Diagnosability," in *Proceedings of the 25th Symposium on the Foundations of Computer Science,* pp. 148-156, 1984.

[TAN 95] TANENBAUM, A. S., *Distributed Operating Systems,* Prentice Hall, 1995.

[UDU 96] UDUPA, D. K., *Network Management Systems Essentials,* McGraw-Hill Series on Computer Communications, 1996.

[YAN 86] YANG, C. -L.; MASSON, G. M., "Hybrid Fault Diagnosability with Unreliable Communication Links," in *Proc. 16th Fault-Tolerant Computing Symp.,* July 1986, pp. 226-231.

APÊNDICE A

Códigos Fontes do Simulador

A.1 Código Principal (*NETSIM.C*)

```
/*-------------------------------------------------------------------*/
/*                          NETSIM.C                                 */
/*                    A simulation program for                       */
/*                Distributed Diagnosis Algorithm                    */
/*                       NetInspector 1.0                            */
/*                          using SMPL                               */
/*-------------------------------------------------------------------*/
/* author:          Moisés Almeida Castelo Branco                    */
/* e-mail:                moises@lia.ufc.br                          */
/*-------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <mem.h>
#include "smpl.h"
#include "netsim.h"

main()
{
  int i,j,r,ntests=0,event,nlks=0;
  int smemsg=0,oldmsg=0,newmsg=0,mixmsg=0,totalmsgs=0,totaltests=0;
  node n;
  struct linktest *t;

  smpl(0,"NetInspector 1.0 Simulation"); /* Inicia o simulador */

  Build_Network();           /* Constrói a rede de interligação dos nós */

  for(i=0;i<NNODES;i++)      /* NNODES = Numero de nós da rede */
    for(j=0;j<NNODES;j++)
      if(Network[i][j])
      {
        Test[ntests].tester=i;
        Test[ntests].tested=j;
        Test[ntests++].lastresult=0;
      }

  for(i=0;i<NNODES;i++)      /* Cria uma facilidade para cada nó */
    Node[i] = facility("Node",1);

  for(i=0;i<NLINKS;i++)
    if (Connection[i])       /* Cria uma facilidade para cada enlace */
    {
      Link[i] = facility("Link",1);
      nlks++;
    }
    else
      Link[i] = -1;
```

```
 for(i=0;i<ntests;i++)     /* Escalona testes iniciais */
  schedule(TEST,0.0,i+1);

/*
   Nesta seção escalonam-se os eventos
   de falha ou reparação de nós e enlaces
*/

/* Eventos de falha e reparação
   ilustrados pelas Figuras 4.13 a 4.18 */

#ifdef EXPERIMENTAL
schedule(NODEFAIL,2.0,1);
schedule(LINKFAIL,15.0,Link_Index(4,5) + 1);
schedule(LINKFAIL,35.0,Link_Index(2,3) + 1);
schedule(NODEREPAIR,43.0,1);
schedule(LINKREPAIR,68.0,Link_Index(4,5) + 1);
schedule(LINKREPAIR,77.0,Link_Index(2,3) + 1);
#endif

/* Inicio do processo de simulação */

j=0;
while(j++<100)      /* Loop principal do simulador */
{
  cause(&event,&i); /* Causa a ocorrência do próximo evento */
  i--;

  switch(event)
  {
     /* Testa o enlace ate o vizinho */

      case TEST:    /* TESTE */

           t=&Test[i];

           /* Verifica se o nó que deve realizar o teste esta normal */

           if (!Test_Node(t->tester))
           {
               totaltests++;
               r=Test_Link(t->tester,t->tested);

               printf("- nó %d testando %d no tempo: %lf\n",
                      t->tester,t->tested, ftTime());

               /* Teste Falhou */
               if (r)
               {

                  /* Verifica novo evento de falha */

                  if (r!=t->lastresult)
                  {

                    printf("* nó %d testando %d - EVENTO DE FALHA DETECTADO\n",
                           t->tester,t->tested);

                    if (Node_Data[t->tester].event_counters[t->tested]%2 == 0)
                       Node_Data[t->tester].event_counters[t->tested]++;

                    /*  Envia mensagens para todos os seus vizinhos */

                    for(n=0;n<NNODES;n++)
                     if (Network[t->tester][n])
                        SendMessage(t->tester,n,NEWMSG,0);

                    t->lastresult = r;
                  }
               }
               else /* Teste passou */
```

```
          {

             /* Verifica novo evento de reparação */

             if (r!=t->lastresult)
             {

                 printf("* nó %d testando %d - EVENTO DE REPARAÇÃO DETECTADO\n",
                        t->tester,t->tested);

                /* Envia uma mensagem apenas para o nó recém reparado */

                SendMessage(t->tester,t->tested,REPMSG,0);

                t->lastresult = r;
             }
          }
       }
       schedule(TEST,10.0,i+1); /* Escalona o próximo teste */
       break;


   /* Recebe uma mensagem de um vizinho */

   case RECEIVE: /* RECEBER_MENSAGEM */

       /* Verifica se o nó destino ou o enlace entre ele e
          o nó originador da mensagem estão falhos */

       if(!Test_Link(Msginfo[i]->from,Msginfo[i]->to))
       {
          totalmsgs++;
          /* Compara Mensagem Com Informação Local */
          switch(CompareInfo(Node_Data[Msginfo[i]->to].event_counters,
                             Msginfo[i]->event_counters))
          {
            case SAMEINFO: /* Mensagem traz informação igual à local */

                 ShowMessage(Msginfo[i]->from,Msginfo[i]->to,SAMEINFO,i,ftTime());
                 smemsg++;
                 break;

            case OLDINFO: /* Mensagem traz apenas informação desatualizada */

                 ShowMessage(Msginfo[i]->from,Msginfo[i]->to,OLDINFO,i,ftTime());
                 oldmsg++;
                 SendMessage(Msginfo[i]->to,Msginfo[i]->from,NEWMSG,0);
                 break;

            case NEWINFO: /* Mensagem traz nova informação */

                 ShowMessage(Msginfo[i]->from,Msginfo[i]->to,NEWINFO,i,ftTime());
                 newmsg++;

                 for(j=0;j<NNODES;j++)
                   Node_Data[Msginfo[i]->to].event_counters[j] =
                   MAX(Node_Data[Msginfo[i]->to].event_counters[j],
                       Msginfo[i]->event_counters[j]);

                 if (Node_Data[Msginfo[i]->to].event_counters[Msginfo[i]->to]%2)
                 {
                    /* Mensagem deve ser atualizada */
                    Node_Data[Msginfo[i]->to].event_counters[Msginfo[i]->to]++;
                    for(n=0;n<NNODES;n++)
                      if (Network[Msginfo[i]->to][n])
                          SendMessage(Msginfo[i]->to,n,NEWMSG,0);
                 }
                 else /* Mensagem deve apenas ser propagada adiante */
                 {
                    for(n=0;n<NNODES;n++)
                      if (Network[Msginfo[i]->to][n] && !(Msginfo[i]->visited[n]))
                          SendMessage(Msginfo[i]->to,n,FWDMSG,i);
```

```
                    }
                    break;

               case MIXEDINFO: /* Mensagem traz informação mista: antiga e nova */

                    ShowMessage(Msginfo[i]->from,Msginfo[i]->to,MIXEDINFO,i,ftTime());
                    mixmsg++;

                    /* A mensagem sempre e atualizada */
                    for(j=0;j<NNODES;j++)
                      Node_Data[Msginfo[i]->to].event_counters[j] =
                      MAX(Node_Data[Msginfo[i]->to].event_counters[j],
                          Msginfo[i]->event_counters[j]);

                    if (Node_Data[Msginfo[i]->to].event_counters[Msginfo[i]->to]%2)
                        Node_Data[Msginfo[i]->to].event_counters[Msginfo[i]->to]++;

                    for(n=0;n<NNODES;n++)
                      if (Network[Msginfo[i]->to][n])
                         SendMessage(Msginfo[i]->to,n,NEWMSG,0);
                    break;

            }
          }
          DisposeMsgEntry(i);
          break;

    /* Evento de falha em um nó */

    case NODEFAIL:  /* FALHA_DE_NÓ */

                    /* Facilidade correspondente requisitada */
                    printf("* falha ocorrida no nó %d no tempo: %lf\n", i, ftTime());
                    request(Node[i],0,0);
                    break;

    /* Evento de reparação de um nó */

    case NODEREPAIR:/* REPARAÇÃO_DE_NÓ */

                    /* Facilidade correspondente liberada */
                    printf("* reparação ocorrida no nó %d no tempo: %lf\n",i,ftTime());
                    release(Node[i],0,0);
                    break;

    /* Evento de falha em um enlace */

    case LINKFAIL:  /* FALHA_DE_ENLACE */

                    /* Facilidade correspondente requisitada */
                    printf("* falha ocorrida no enlace %d no tempo: %lf\n",i,ftTime());
                    request(Link[i],0,0);
                    break;

    /* Evento de reparação de um enlace */

    case LINKREPAIR:/* REPARAÇÃO_DE_ENLACE */

                    /* Facilidade correspondente liberada */
                    printf("* reparação ocorrida no enlace %d no tempo: %lf\n",
                            i,ftTime());
                    release(Link[i],0,0);
                    break;
  }
}

Debug();

for(i=0;i<NNODES;i++)
{
  for(j=0;j<NNODES;j++)
    printf("%d ",Node_Data[i].event_counters[j]);
```

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

  printf("\n");
  printf("ESTATÍSTICAS DE MENSAGENS:\n\n");
  printf("MESMAINFO  : %d\n",smemsg);
  printf("ANTIGAINFO : %d\n",oldmsg);
  printf("NOVAINFO   : %d\n",newmsg);
  printf("MISTAINFO  : %d\n",mixmsg);
  printf("      TOTAL: %d\n\n",totalmsgs);
  printf("TOTAL DE TESTES    : %d\n",totaltests);
  printf("TOTAL DE ENLACES   : %d\n",nlks);

/*  reportf(); */

}

void Build_Network() /* Constrói a rede de interligação dos nós */
{
  node i,j;
  short link;

  memset(Network,0,NNODES*NNODES*sizeof(node));
  memset(Connection,0,NLINKS*sizeof(short));
  randomize();
  for (i=0;i<NNODES;i++)
    for (j=i+1;j<NNODES;j++) {

#ifdef FULLCONNECTED              /* REDE COMPLETAMENTE CONECTADA */
       Network[i][j] = 1;
#else
       Network[i][j] = random(2); /* REDE ALEATÓRIA */
#endif

       if(Network[i][j])
       {
        Connection[Link_Index(i,j)] = 1;
        Network[j][i] = Network[i][j];
       }
    }

#ifdef SINGLEPATH                /* REDE ESPARSA */
  memset(Network,0,NNODES*NNODES*sizeof(node));
  memset(Connection,0,NLINKS*sizeof(short));
#endif

  /* Impede geração de rede desconexa */
  for (i=0;i<NNODES;i++) {
    if(i<NNODES-1) {
      if(!Network[i][i+1]) {
       Network[i][i+1] = 1;
       Network[i+1][i] = 1;
       Connection[Link_Index(i,i+1)] = 1;
      }
    }
    else {
      int n;
      n = random(NNODES);
      if (n!=i) {
       Network[i][n] = 1;
       Network[n][i] = 1;
       if (n>i)
         Connection[Link_Index(i,n)] = 1;
       else
         Connection[Link_Index(n,i)] = 1;
      }
    }
  }
}

short Link_Index(node i, node j)
{
```

```
  int k,sum=0;

  for(k=0;k<i;k++)
    sum+=NNODES-k-1;

  return (j-i-1) + sum;
}

short Test_Link(node i,node j)
{
  if (i<j)
    return (status(Link[Link_Index(i,j)])  |
            status(Node[j])) ? FAILED : PASSED;
  else
    return (status(Link[Link_Index(j,i)])  |
            status(Node[j])) ? FAILED : PASSED;
}

short Test_Node(node i)
{
  return status(Node[i]) ? FAILED : PASSED;
}

short GetFreeMsgEntry(void)
{
  short i,next=-1;

  for(i=0;i<MAXMSGS;i++)
   if (Msginfo[i]==NULL) {
     if ((Msginfo[i] = (struct message *)
                       malloc(sizeof(struct message)))==NULL) {
       printf("ERRO: NÃO FOI POSSÍVEL ALOCAL MEMÓRIA PARA MENSAGEM.\n");
       exit(0);
     }
     next=i;
     break;
   }

   if(next==-1) {
     printf("ERRO: NÚMERO MÁXIMO DE MENSAGENS EXCEDIDO.\n");
     exit(0);
   }

   return next;
}

void DisposeMsgEntry(int entry)
{
  free(Msginfo[entry]);
  Msginfo[entry] = NULL;
}

short CompareInfo(node *SelfInfo,node *MsgInfo)
{
  short msghasoldinfo=0,msghasnewinfo=0,k;

  for(k=0;k<NNODES;k++)
  {
    if(SelfInfo[k] < MsgInfo[k])
      msghasnewinfo = 1;

    if(SelfInfo[k] > MsgInfo[k])
      msghasoldinfo = 1;
  }

  if(msghasnewinfo && msghasoldinfo)
    return MIXEDINFO;

  if(msghasnewinfo)
    return NEWINFO;

  if(msghasoldinfo)
```

```
    return OLDINFO;

  return SAMEINFO;
}


void Debug()
{
  int i,j;

  for (i=0;i<NNODES;i++) {
    for (j=0;j<NNODES;j++)
      printf("%c%c",Network[i][j] ? 178:176,Network[i][j] ? 178:176);
    printf("\n");
  }
}

void SendMessage(node from,node to,short type,short entry)
{
   int m,n;

   m = GetFreeMsgEntry ();
   Msginfo[m]->from = from;
   Msginfo[m]->to   = to;
   memcpy(Msginfo[m]->event_counters,
          Node_Data[from].event_counters,
          NNODES*sizeof(node));

   switch(type)
   {
     /* MENSAGEM ENVIADA QUANDO DETECTADA A REPARAÇÃO DE UM NÓ OU ENLACE */
     case REPMSG: memset(Msginfo[m]->visited,0,NNODES*sizeof(node));
                  Msginfo[m]->visited[from] = 1;
                  break;

     /* MENSAGEM NOVA */
     case NEWMSG: memcpy(Msginfo[m]->visited,Network[from],NNODES*sizeof(node));
                  Msginfo[m]->visited[from] = 1;
                  break;

     /* MENSAGEM PARA SER PROPAGADA ADIANTE */
     case FWDMSG: memcpy(Msginfo[m]->visited,Network[from],NNODES*sizeof(node));
                  for(n=0;n<NNODES;n++)
                  if (!Msginfo[m]->visited[n] && Msginfo[entry]->visited[n])
                     Msginfo[m]->visited[n] = 1;
                  Msginfo[m]->visited[from] = 1;
                 break;
   }

   schedule(RECEIVE,1.0,m+1);
}

void ShowMessage(node from, node to, short type, short entry, double time)
{
  int i;

  switch(type)
  {
    case SAMEINFO:  printf("MESMAInfo  recebida no nó %d de %d no tempo %lf\n",
                           to,from,time);
                    break;
    case OLDINFO:   printf("ANTIGAInfo recebida no nó %d de %d no tempo %lf\n",
                           to,from,time);
                    break;
    case NEWINFO:   printf("NOVAInfo   recebida no nó %d de %d no tempo %lf\n",
                           to,from,time);
                    break;
    case MIXEDINFO: printf("MISTAInfo  recebida no nó %d de %d no tempo %lf\n",
                           to,from,time);
                    break;
  }
```

```
  printf("contadores de eventos =>");
  for(i=0;i<NNODES;i++)
     printf("  %d",Msginfo[entry]->event_counters[i]);
  printf("\n");
}
```

A.2 Arquivo *header* (*NETSIM.H*)

```
/*--------------------------------------------------------------------*/
/*                            NETSIM.H                                */
/*                  header file for NETSIM.C                          */
/*--------------------------------------------------------------------*/
/* author:          Moises Almeida Castelo Branco                     */
/* e-mail:               moises@lia.ufc.br                            */
/*--------------------------------------------------------------------*/

typedef unsigned short node;

/* Lista de eventos */
#define TEST         1
#define RECEIVE      2
#define NODEFAIL     3
#define NODEREPAIR   4
#define LINKFAIL     5
#define LINKREPAIR   6

#define FAILED   1
#define PASSED   !FAILED

#define MAXMSGS  1000 /* Numero máximo de mensagens */

/* Resultados de comparação da mensagem com informação local */
#define SAMEINFO   100
#define OLDINFO    200
#define NEWINFO    300
#define MIXEDINFO  400

/* Tipos de propagação de mensagem */
#define REPMSG 20 /* Mensagem enviada no retorno de um nó ou enlace */
#define NEWMSG 30 /* Mensagem nova (recém criada ou alterada) */
#define FWDMSG 40 /* Mensagem propagada adiante (não alterada) */

#ifdef EXPERIMENTAL
#define NNODES 7 /* Numero de nos da rede experimental das Figuras 4.13 a 4.18 */
#else
#define NNODES 3 /* Numero de nós da rede */
#endif

#define NLINKS ((NNODES*(NNODES-1))/2) /* Numero de enlaces possíveis */

#define MAX(A,B) ((A) >= (B)) ? (A):(B)

/* Estrutura de dados para o nó */
struct node_data
{
  node event_counters[NNODES];
} Node_Data[NNODES];

/* Estrutura de dados para um teste */
struct linktest
{
  node tester;
  node tested;
  short lastresult;
} Test[2*NLINKS];

/* Estrutura de dados para mensagem */
```

```
struct message
{
  node from;
  node to;

  node visited[NNODES];
  node event_counters[NNODES];
} *Msginfo[MAXMSGS];

#ifndef EXPERIMENTAL
node Network[NNODES][NNODES]; /* Estrutura de dados com a rede de interligação dos nós */
#else
/* Rede experimental usada para simular os exemplos das Figuras 4.13 a 4.18 */
node Network[7][7] = { {0,1,1,0,0,0,0},
                       {1,0,1,0,0,0,0},
                       {1,1,0,1,0,0,0},
                       {0,0,1,0,1,0,1},
                       {0,0,0,1,0,1,0},
                       {0,0,0,0,1,0,1},
                       {0,0,0,1,0,1,0} };
#endif

short Connection[NLINKS];     /* Estrutura de dados com os enlaces */
node Node[NNODES];            /* Facilidades para cada nó */
short Link[NLINKS];           /* Facilidades para cada enlace */

/* Protótipos das funções */
void Debug();
void Build_Network();
short Link_Index(node i,node j);
short Test_Link (node i,node j);
short Test_Node (node i);
short GetFreeMsgEntry(void);
void DisposeMsgEntry(int entry);
short CompareInfo(node *SelfInfo,node *MsgInfo);
void SendMessage(node from, node to, short type, short entry);
void ShowMessage(node from, node to, short type, short entry,double time);
```

APÊNDICE B

Códigos Fontes do Programa NetInspector

B.1 Código Principal (*NetInspect.java*)

```
/*-----------------------------------------------------------------*/
/*                       NetInspector 1.0                          */
/*      A Fault Diagnosis Program for General Topology Networks    */
/*                                                                 */
/*                         Java Version                            */
/*                                                                 */
/*                    implemented using JDK1.1.4                   */
/*                       www.java.sun.com                          */
/*-----------------------------------------------------------------*/
/* author:          Moisés Almeida Castelo Branco                  */
/* e-mail:                moises@lia.ufc.br                        */
/*-----------------------------------------------------------------*/
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class NetInspect implements ActionListener // Classe Principal
{
   static Frame mainwindow;
   static TextArea textwindow;
   static Button diagnosis;
   static Button clear;
   static Button exit;
   static Button info;
   Panel pCnt;

   static TestsServer testsServerProc;
   static DataServer  dataServerProc;
   static final int TestsServerPort = 5000; // Porta do Servidor de Testes.
   static final int DataServerPort  = 4000; // Porta do Servidor de Mensagens.
   static int MessageSize;

   static int Counters[];
   static byte VisitedAux[];

   static int NodeId[];
   static String IPNumber[], HostName[], HostNickName[];
   static int NCurrentNodes=0, NCurrentNeighbors=0;
   static final int NMAXNODES = 100;

   static int WhoAmI;
   static int MyNeighbors[];

   static boolean debug=false;

   public static void main(String[] args) throws IOException
   {

      // Lê os Arquivos de Configurações do Programa.

      // SYSHOSTS.INF - ARQUIVO COM OS ENDEREÇOS DE TODOS OS HOSTS PARTICIPANTES DO
      //                SISTEMA.
```

```
    // NODECFG.INF  - ARQUIVO COM INFORMAÇÃO SOBRE O PRÓPRIO NÓ.
    ReadConfigurationFiles.ReadFiles("syshosts.inf","nodecfg.inf");

    new NetInspect();

    Counters = new  int[NCurrentNodes]; // CONTADORES DE EVENTOS LOCAIS
    MessageSize = 5*NCurrentNodes + 4;

    try
    {
      console("Iniciando servidor de testes....");
      testsServerProc = new TestsServer(TestsServerPort);
      testsServerProc.start();
      console("Servidor iniciado com sucesso.");
    }
    catch (Exception e)
    {
      console ( e.toString() );
    }

    try
    {
      console("Iniciando servidor de dados....");
      dataServerProc = new DataServer(DataServerPort);
      dataServerProc.start();
      console("Servidor iniciado com sucesso.");
    }
    catch (Exception e)
    {
      console ( e.toString() );
    }

    for(int k=0;k<NCurrentNeighbors;k++)
        new DoTestThread(MyNeighbors[k],TestsServerPort,3000).start();


    // Envia mensagem com contadores nulos para todos os vizinhos
    // ao iniciar ou ter sido reparado.
    for(int k=0;k<NCurrentNodes;k++)
        VisitedAux[k] = 0;
    for(int k=0;k<NCurrentNeighbors;k++)
        VisitedAux[MyNeighbors[k]] = 1;

    VisitedAux[WhoAmI] = 1;

    for(int i=0;i<NCurrentNeighbors;i++)
        new SendMessageThread(MyNeighbors[i],DataServerPort,
                              VisitedAux,Counters).start();


}

public NetInspect () {

  VisualInterface();
  EventHandlers();

}

public void VisualInterface() { // Interface Gráfica com Usuário.

  mainwindow = new Frame( "NetInspector 1.0 * Diagnostico de Falhas em Redes de
                           Topologia Geral." );
  mainwindow.setLocation( 50, 50 );
  mainwindow.setSize( 350, 350 );

  textwindow = new TextArea();
  textwindow.setEditable( false );

  diagnosis = new Button( "Diagnostico" );
  info      = new Button( "Informação Geral" );
```

```
    clear     = new Button( "Limpar Console de Mensagens" );
    exit      = new Button( "Sair" );

    pCnt = new Panel( new GridLayout( 4, 1 ) );
    pCnt.add( diagnosis );
    pCnt.add( info );
    pCnt.add( clear );
    pCnt.add( exit );

    mainwindow.add( "North",  new Label( "Mensagens:" ) );
    mainwindow.add( "Center", textwindow );
    mainwindow.add( "South", pCnt );

    mainwindow.addWindowListener(
       new WindowAdapter() {
          public void windowClosing( WindowEvent event )
          {
             console( "Fechando Servidores..." );
             try {
                   testsServerProc.stop();
                   dataServerProc.stop();
                   console( "Finalizado." );
                   System.exit( 0 );
                  }
             catch( IOException e ) {
                   console( e.toString() );
                   System.exit( 1 );
                  }
             }
           }
    );
    mainwindow.setVisible( true );
}

public void EventHandlers ()
{
    diagnosis.addActionListener( this );
    clear.addActionListener( this );
    info.addActionListener( this );
    exit.addActionListener( this );
}

public void actionPerformed( ActionEvent event )
{
    String command;

    try
    {
       command = event.getActionCommand();

       if (command.equalsIgnoreCase("diagnostico"))
       {
          console("Diagnostico:\n");
          for(int k=0;k<NCurrentNodes;k++)
            console(IPNumber[k] + " (" + HostNickName[k] + ") " + " ---> estado: " +
                    ((Counters[k]%2==1) ? "FALHA" : "NORMAL"));

       }

       if (command.equalsIgnoreCase("informação geral"))
       {
          console("-------------------------------------");
          console("Hosts do Sistema:\n");
          for(int k=0;k<NCurrentNodes;k++)
          {
             console("Id: " + NodeId[k] +
                     "\nIPNumber: " + IPNumber[k] +
                     "\nHostName: " + HostName[k] +
                     "\nHostNick: " + HostNickName[k]);
          }

          console("-------------------------------------");
```

```
            console("Informação Própria:\n");
            console("Id: " + NodeId[WhoAmI] +
                    "\nIPNumber: " + IPNumber[WhoAmI] +
                    "\nHostName: " + HostName[WhoAmI] +
                    "\nHostNick: " + HostNickName[WhoAmI]);

            console("-----------------------------------");
            console("Informação Sobre os Vizinhos:\n");
            for(int k=0;k<NCurrentNeighbors;k++)
               console("Neighbor("+(k+1)+"): " +
                        "\nId: " + NodeId[MyNeighbors[k]] +
                        "\nIPNumber: " + IPNumber[MyNeighbors[k]] +
                        "\nHostName: " + HostName[MyNeighbors[k]] +
                        "\nHostNick: " + HostNickName[MyNeighbors[k]]);
            console("-----------------------------------");
         }

         if (command.equalsIgnoreCase("limpar console de mensagens"))
         {
            textwindow.setText(" ");
         }

         if (command.equalsIgnoreCase("sair"))
         {
            console( "Fechando Servidores..." );
            try {
                  testsServerProc.stop();
                  dataServerProc.stop();
                  console( "Finalizado." );
                  System.exit( 0 );
                 }
            catch( IOException e ) {
                  console( e.toString() );
                  System.exit( 1 );
            }
         }
      }
      catch( Exception e )
      {
         console( e.toString() );
      }
   }

   public static void console( String msg )
   {
      textwindow.append( msg + "\n" );
   }

}

class TestsServer implements Runnable // Servidor de testes.
{
    Thread       TestsServerThread = null;
    ServerSocket TestsServerSocket = null;
    int Port;

    public TestsServer(int port) throws IOException
    {
      Port = port;
    }

    public synchronized void start() throws IOException
    {
        if ( TestsServerSocket == null )
        {
           TestsServerSocket = new ServerSocket( Port );
        }
        if ( TestsServerThread == null )
        {
            TestsServerThread = new Thread( this );
            TestsServerThread.setPriority( Thread.MAX_PRIORITY / 4 );
            TestsServerThread.start();
```

```
        }
    }

    public synchronized void stop() throws IOException
    {
        if ( TestsServerSocket != null )
            TestsServerSocket.close();

        if ( TestsServerThread != null )
            TestsServerThread.stop();
    }

    public void run()
    {
       while(true)
       {
          try
          {
            new TestReplyThread(TestsServerSocket.accept()).start();
          }
          catch (IOException e)
          {
            NetInspect.console( e.toString() );
          }
       }
    }
}

class TestReplyThread extends Thread // Responde a um teste.
{
  private Socket socket = null;

  public TestReplyThread(Socket socket)
  {
    super("TestReplyThread");
    this.socket = socket;
  }

  public void run ()
  {
      InputStream          IncomingTest;
      BufferedOutputStream TestReply;

      try
      {
           IncomingTest = new BufferedInputStream ( socket.getInputStream () );
           TestReply    = new BufferedOutputStream( socket.getOutputStream() );

           int nbytes;
           byte b[] = new byte[2];

           if ( (nbytes = IncomingTest.read( b, 0 , 2)) == 2)
           {
              TestReply.write( b[0] );
              TestReply.write( b[1] );
           }

           TestReply.flush();
           TestReply.close();

           IncomingTest.close();
           socket.close();
      }
      catch (IOException e)
      {
        NetInspect.console( e.toString() );
      }
  }
}

class DoTestThread extends Thread // Realiza um teste em um vizinho.
{
```

```
private int Neighbor;
private int Port;
private int Interval;
boolean LastTestFailed = false;

byte Visited[] = new byte[NetInspect.NCurrentNodes];

public DoTestThread(int neighbor, int port, int interval)
{
  super("DoTestThread");
  Neighbor = neighbor;
  Port     = port;
  Interval = interval;
}

public void run()
{
  BufferedOutputStream Test;
  InputStream          Reply;
  Socket               socket=null;

  while(true)
  {
     try
     {
         socket = new Socket(NetInspect.IPNumber[Neighbor], Port) ;

         Test  = new BufferedOutputStream( socket.getOutputStream() );
         Reply = new BufferedInputStream ( socket.getInputStream () );

         byte TestType='1', TestValue='0';

         Test.write( TestType  );
         Test.write( TestValue );

         Test.flush();

         int nbytes;
         byte b[] = new byte[2];

         if ( (nbytes = Reply.read( b, 0 , 2)) == 2)
         {
            if (b[0]!=TestType || b[1]!=TestValue)
              NetInspect.console("Teste falhou.");
            else
            {
              if (LastTestFailed)
              {
                LastTestFailed = false;

                for(int k=0;k<NetInspect.NCurrentNodes;k++)
                    Visited[k] = 0;
                Visited[NetInspect.WhoAmI] = 1;

                new SendMessageThread(Neighbor,NetInspect.DataServerPort,
                                      Visited,NetInspect.Counters).start();

              }
            }
         }

         Test.close();
         Reply.close();
         socket.close();

     }
     catch (IOException e)
     {
       if (!LastTestFailed)
       {
         LastTestFailed = true;
         if (NetInspect.Counters[Neighbor]%2 == 0)
```

```
               NetInspect.Counters[Neighbor]++;

            for(int k=0;k<NetInspect.NCurrentNodes;k++)
                Visited[k] = 0;
            for(int k=0;k<NetInspect.NCurrentNeighbors;k++)
                Visited[NetInspect.MyNeighbors[k]] = 1;
            Visited[NetInspect.WhoAmI] = 1;

            for(int i=0;i<NetInspect.NCurrentNeighbors;i++)
              new SendMessageThread(NetInspect.MyNeighbors[i],NetInspect.DataServerPort,
                                    Visited,NetInspect.Counters).start();
          }
        }
        try {
          sleep(Interval);
        }
        catch (InterruptedException e)
        {
          NetInspect.console( e.toString() );
        }
    }
  }
}

class ByteToInt{ // Converte quatro bytes em um tipo inteiro.

    public static int bytetoInt(byte b1, byte b2, byte b3, byte b4)
    {
      int l,i1,i2,i3,i4;

      i1=(b1>=0) ? b1: 256 + b1;
      i2=(b2>=0) ? b2: 256 + b2;
      i3=(b3>=0) ? b3: 256 + b3;
      i4=(b4>=0) ? b4: 256 + b4;

      l = (i1 << 24) | (i2 << 16) + (i3 << 8) + i4;

      return l;
    }
}

class DataServer implements Runnable // Recebe uma mensagem de um vizinho.
{
    Thread       DataServerThread = null;
    ServerSocket DataServerSocket = null;
    int Port;

    byte MsgVisited[] = new byte[NetInspect.NCurrentNodes];
    byte FwdVisited[] = new byte[NetInspect.NCurrentNodes];
    int MsgCounters[] = new int [NetInspect.NCurrentNodes];

    int Sender;

    public DataServer(int port) throws IOException
    {
      Port = port;
    }

    public synchronized void start() throws IOException
    {
        if ( DataServerSocket == null )
        {
           DataServerSocket = new ServerSocket( Port );
        }
        if ( DataServerThread == null )
        {
            DataServerThread = new Thread( this );
            DataServerThread.setPriority( Thread.MAX_PRIORITY / 4 );
            DataServerThread.start();
        }
    }
```

```
public synchronized void stop() throws IOException
{
    if ( DataServerSocket != null )
        DataServerSocket.close();

    if ( DataServerThread != null )
        DataServerThread.stop();
}

public void run()
{
   InputStream          IncomingData;
   Socket socket;

   while(true)
   {
      try
      {
         socket = DataServerSocket.accept();

         IncomingData = new BufferedInputStream ( socket.getInputStream () );

         int nbytes;
         byte b[] = new byte[NetInspect.MessageSize];

         if ( (nbytes = IncomingData.read( b, 0 , NetInspect.MessageSize)) ==
               NetInspect.MessageSize)
         {
            Sender = ByteToInt.bytetoInt(b[0],b[1],b[2],b[3]);

            NetInspect.console("Mensagem Recebida de : " +
                   socket.getInetAddress() + " NodeId: " + Sender);

            System.arraycopy(b,4,MsgVisited,0,NetInspect.NCurrentNodes);

            for(int i=0;i<NetInspect.NCurrentNodes;i++)
              System.out.print(MsgVisited[i] +  " ");

            NetInspect.console("\n");

            for(int k=NetInspect.NCurrentNodes+4;k<NetInspect.MessageSize;k+=4)
            {
             MsgCounters[(k-(NetInspect.NCurrentNodes+4))/4] =
             ByteToInt.bytetoInt(b[k],b[k+1],b[k+2],b[k+3]);
             System.out.print(MsgCounters[(k-(NetInspect.NCurrentNodes+4))/4] + " ");
            }

            int info = CompareInformation.CompareInfo(NetInspect.Counters,
                                                      MsgCounters);

            switch(info)
            {
              case 0: /* MESMAInfo */
                      break;

              case 1: /* ANTIGAInfo  */
                      for(int k=0;k<NetInspect.NCurrentNodes;k++)
                          MsgVisited[k]=0;
                      MsgVisited[NetInspect.WhoAmI]=1;
                      new SendMessageThread(Sender,
                                            NetInspect.DataServerPort,
                                            MsgVisited,
                                            NetInspect.Counters).start();
                      break;

              case 2: /* NOVAInfo  */
                      for(int k=0;k<NetInspect.NCurrentNodes;k++)
                         NetInspect.Counters[k] = (NetInspect.Counters[k] >=
                                                   MsgCounters[k]) ?
                                                   NetInspect.Counters[k] :
                                                   MsgCounters[k];
```

```
                    if (NetInspect.Counters[NetInspect.WhoAmI] % 2 == 1)
                    {
                        NetInspect.Counters[NetInspect.WhoAmI]++;

                        for (int k=0;k<NetInspect.NCurrentNodes;k++)
                             MsgVisited[k] = 0;
                        for (int k=0;k<NetInspect.NCurrentNeighbors;k++)
                             MsgVisited[NetInspect.MyNeighbors[k]] = 1;
                        MsgVisited[NetInspect.WhoAmI] = 1;

                        for(int k=0;k<NetInspect.NCurrentNeighbors;k++)
                            new SendMessageThread(NetInspect.MyNeighbors[k],
                                                  NetInspect.DataServerPort,
                                                  MsgVisited,
                                                  NetInspect.Counters).start();
                    }
                    else
                    {
                        for (int k=0;k<NetInspect.NCurrentNodes;k++)
                             FwdVisited[k] = MsgVisited[k];
                        for (int k=0;k<NetInspect.NCurrentNeighbors;k++)
                             FwdVisited[NetInspect.MyNeighbors[k]] = 1;
                        FwdVisited[NetInspect.WhoAmI] = 1;

                        for(int k=0;k<NetInspect.NCurrentNeighbors;k++)
                          if (MsgVisited[NetInspect.MyNeighbors[k]] == 0)
                            new SendMessageThread(NetInspect.MyNeighbors[k],
                                                  NetInspect.DataServerPort,
                                                  FwdVisited,
                                                  NetInspect.Counters).start();

                    }
                    break;

             case 3: /* MISTAInfo */
                    for(int k=0;k<NetInspect.NCurrentNodes;k++)
                       NetInspect.Counters[k] = (NetInspect.Counters[k] >=
                                                 MsgCounters[k]) ?
                                                 NetInspect.Counters[k] :
                                                 MsgCounters[k];

                    if (NetInspect.Counters[NetInspect.WhoAmI] % 2 == 1)
                        NetInspect.Counters[NetInspect.WhoAmI]++;

                    for (int k=0;k<NetInspect.NCurrentNodes;k++)
                         MsgVisited[k] = 0;
                    for (int k=0;k<NetInspect.NCurrentNeighbors;k++)
                         MsgVisited[NetInspect.MyNeighbors[k]] = 1;
                    MsgVisited[NetInspect.WhoAmI] = 1;

                    for(int k=0;k<NetInspect.NCurrentNeighbors;k++)
                       new SendMessageThread(NetInspect.MyNeighbors[k],
                                             NetInspect.DataServerPort,
                                             MsgVisited,
                                             NetInspect.Counters).start();

                    break;
               }

               NetInspect.console("\n");
            }
            else
            {
               NetInspect.console("Mensagem Invalida Recebida de : " +
                      socket.getInetAddress() );
            }

            IncomingData.close();
            socket.close();
        }
        catch (IOException e)
        {
```

```
            NetInspect.console( e.toString() );
          }
        }
    }
}

class SendMessageThread extends Thread // Envia uma mensagem a um vizinho.
{
  private int Neighbor;
  private int Port;
  private byte Visited[];
  private int Counters[];

  public SendMessageThread(int neighbor, int port, byte visited[], int counters[])
  {
    super("SendMessageThread");
    Neighbor = neighbor;
    Port     = port;
    Visited  = visited;
    Counters = counters;
  }

  public void run()
  {
    BufferedOutputStream Message;
    Socket               socket=null;

    try
    {
        socket = new Socket(NetInspect.IPNumber[Neighbor], Port);

       Message     = new BufferedOutputStream( socket.getOutputStream() );

        int p=255;

        // Quebra o Tipo Inteiro nos Quatro Bytes que o Compõem.
        Message.write( (byte) (NetInspect.WhoAmI >> 24 & p) );
        Message.write( (byte) (NetInspect.WhoAmI >> 16 & p) );
        Message.write( (byte) (NetInspect.WhoAmI >> 8 & p) );
        Message.write( (byte) (NetInspect.WhoAmI & p) );

       for(int k=0;k<Visited.length;k++)
          Message.write(Visited[k]);

        for(int k=0;k<Counters.length;k++)
       {
          // Quebra o Tipo Inteiro nos Quatro Bytes que o Compõem.
          Message.write( (byte) (Counters[k] >> 24 & p) );
          Message.write( (byte) (Counters[k] >> 16 & p) );
          Message.write( (byte) (Counters[k] >> 8 & p) );
          Message.write( (byte) (Counters[k] & p) );
       }

       Message.flush();

       Message.close();
       socket.close();

    }
    catch (IOException e)
    {
      NetInspect.console( e.toString() );
    }
  }
}

class CompareInformation{  // Compara Informações da Mensagem Com Informação Local.

   public static int CompareInfo(int SelfInfo[], int MsgInfo[])
   {
      boolean msghasoldinfo=false, msghasnewinfo=false;
```

```
      for(int k=0;k<SelfInfo.length;k++)
      {
         if (SelfInfo[k] < MsgInfo[k])
           msghasnewinfo = true;

         if (SelfInfo[k] > MsgInfo[k])
           msghasoldinfo = true;
      }

      if (msghasnewinfo && msghasoldinfo)  // Informação Mista - MISTAInfo
         return 3;
      if (msghasnewinfo) // Informação Nova - NOVAInfo
         return 2;
      if (msghasoldinfo) // Informação Antiga - ANTIGAInfo
         return 1;

      return 0; // Informação Igual - MESMAInfo
   }
}

class ReadConfigurationFiles {  // Lê os Arquivos de Configurações do Programa

   public static void ReadFiles(String syshostsfile, String nodecfgfile) throws
                                                                        IOException
   {
     try {
       DataInputStream in = new DataInputStream(new
       FileInputStream(syshostsfile));

       NetInspect.NodeId       = new int   [NetInspect.NMAXNODES];
       NetInspect.IPNumber     = new String[NetInspect.NMAXNODES];
       NetInspect.HostName     = new String[NetInspect.NMAXNODES];
       NetInspect.HostNickName = new String[NetInspect.NMAXNODES];

       String Token="";
       boolean comment=false;
       int tokenord=0;
       byte c;

       try {
             while(true) {
                c = in.readByte();
                if (c!=' ' && c!='\n')
                {
                   if (c!='\r')
                   Token = Token + (char) c;
                }
                else
                {
                   switch(tokenord++)
                   {
                      case 0:  if (Token.charAt(0) == ';')
                                  comment = true;
                               else
                                  NetInspect.NodeId[NetInspect.NCurrentNodes] =
                                                  Integer.parseInt(Token);
                               break;
                      case 1:  NetInspect.IPNumber[NetInspect.NCurrentNodes] = Token;
                               break;
                      case 2:  NetInspect.HostName[NetInspect.NCurrentNodes] = Token;
                               break;
                      case 3:  NetInspect.HostNickName[NetInspect.NCurrentNodes] = Token;
                               break;
                      default: if(!comment)
                               {
                                  System.out.println("Arquivo Inválido: " +
                                                     syshostsfile);
                                  System.out.println("Informação excedente: " + "'"
                                                     + Token + "'" + " na linha: "
                                                     +(NetInspect.NCurrentNodes+1));
                                  in.close();
                                  System.exit(1);
```

```
                        }
                }
                Token = "";

                if (c=='\n')
                {
                   if ((tokenord!=4 || (NetInspect.NodeId[NetInspect.NCurrentNodes] !=
                                        NetInspect.NCurrentNodes)) && !comment)
                   {
                      System.out.println("Arquivo Invalido: " + syshostsfile);
                      System.out.println("Informações faltando ou incorretas na linha:
                                          " + (NetInspect.NCurrentNodes+1));
                      in.close();
                      System.exit(1);
                   }
                   else
                   {
                      if (!comment)
                        NetInspect.NCurrentNodes++;

                      comment=false;
                      tokenord=0;
                      Token="";
                   }
                }
             }
           }
        } catch (EOFException e) {
        }
        in.close();
     } catch (FileNotFoundException e) {
        System.out.println( "Arquivo não encontrado: " + syshostsfile);
        System.exit(1);
     }

try {

  DataInputStream in = new DataInputStream(new
  FileInputStream(nodecfgfile));

  NetInspect.WhoAmI=0;
  NetInspect.MyNeighbors = new int[NetInspect.NMAXNODES];

  String Token="";
  boolean comment=false, who=false, myn=false;
  int tokenord=0, linenumber=0, neighbors=0;
  byte c;

  try {
        while(true) {
           c = in.readByte();
           if (c!='=' && c!=';' && c!='\n')
           {
              if (c!='\r')
              Token = Token + (char) c;
           }
           else
           {
             tokenord++;

             if(tokenord==1)
             {
               if (Token.equalsIgnoreCase("whoami") && c=='=' && linenumber==0)
               {
                  who=true;
                  Token="";
               }
               else
               {
                 if (Token.equalsIgnoreCase("myneighbors")
                      && c=='=' && linenumber==1)
                 {
```

```
                        myn=true;
                        Token="";
                     }
                     else
                     {
                        System.out.println("Arquivo Inválido: " + nodecfgfile);
                        System.out.println("Erro na linha " + (linenumber+1));
                        in.close();
                        System.exit(1);
                     }
                  }
               }
               else
               {
                  if(who)
                  {
                     NetInspect.WhoAmI=Integer.parseInt(Token);
                     who=false;
                     Token="";
                     tokenord=0;
                     if (c=='\n')
                       linenumber++;
                  }
                  else
                  {
                     if(myn)
                     {
                         NetInspect.MyNeighbors[NetInspect.NCurrentNeighbors] =
                         Integer.parseInt(Token);
                         if (NetInspect.MyNeighbors[NetInspect.NCurrentNeighbors] >
                         NetInspect.NCurrentNodes - 1)
                         {
                           System.out.println("Erro: Nao ha informacoes sobre 'NodeId: "
                           + NetInspect.MyNeighbors[NetInspect.NCurrentNeighbors]
                           + "' no sistema.");
                           System.out.println("Verifique os arquivos " + nodecfgfile + "
                           e " + syshostsfile + ".");
                           in.close();
                           System.exit(1);
                         }
                         NetInspect.NCurrentNeighbors++;
                         Token="";
                         if (c=='\n')
                           linenumber++;
                         if (c!=';')
                           myn=false;
                     }
                     else
                     {
                        if (linenumber!=2)
                        {
                          System.out.println("Arquivo Invalido: " + nodecfgfile);
                          System.out.println("Erro na linha: " + (linenumber+1));
                          in.close();
                          System.exit(1);
                        }
                     }
                  }
               }
            }
         }
        } catch (EOFException e) {
        }
        in.close();
      } catch (FileNotFoundException e) {
        System.out.println( "Arquivo não encontrado: " + nodecfgfile);
        System.exit(1);
      }
   }
}
```

B.2 Sistema de Diagnóstico Real e Arquivos de Configurações

A Figura B.2.1 mostra um sistema exemplo composto de 7 estações (nós) e uma topologia de interligação entre eles.

A Figura B.2.2 mostra o arquivo SYSHOSTS.INF que deve estar disponível em cada uma das estações.

A Figura B.2.3 mostra os arquivos NODECFG.INF para cada uma das estações participantes do sistema de diagnóstico. Estes arquivos determinam a topologia lógica de interligação do sistema de diagnóstico. Para que duas estações sejam vizinhas é necessário que estejam conectadas através de um enlace ponto a ponto (vizinhas físicas) ou *broadcast* (vizinhas lógicas - ex. *Ethernet*).

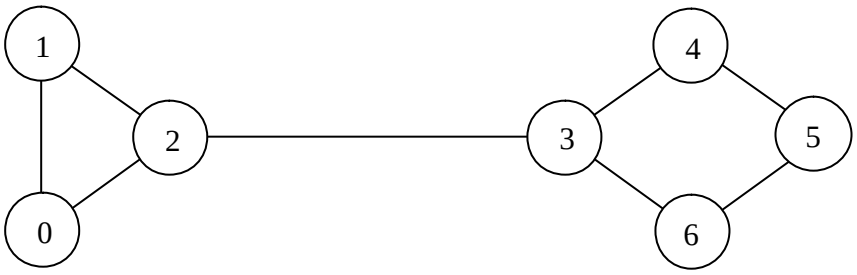


Figura B.2.1 - Rede composta por 7 estações (nós)

```
; Arquivo com todos os HOSTS do sistema de diagnostico
0 10.85.9.102 ce051593.teleceara.com.br ce051593
1 10.85.10.96 ce052683.teleceara.com.br ce052683
2 10.85.10.27 ce051685.teleceara.com.br ce051685
3 200.19.177.4 iracema.lia.ufc.br iracema
4 200.19.177.12 taiba.lia.ufc.br taiba
5 200.19.177.6 iguape.lia.ufc.br iguape
6 200.19.177.5 paracuru.lia.ufc.br paracuru
```

Figura B.2.2 - Conteúdo do arquivo SYSHOSTS.INF para a rede da Figura B.2.1

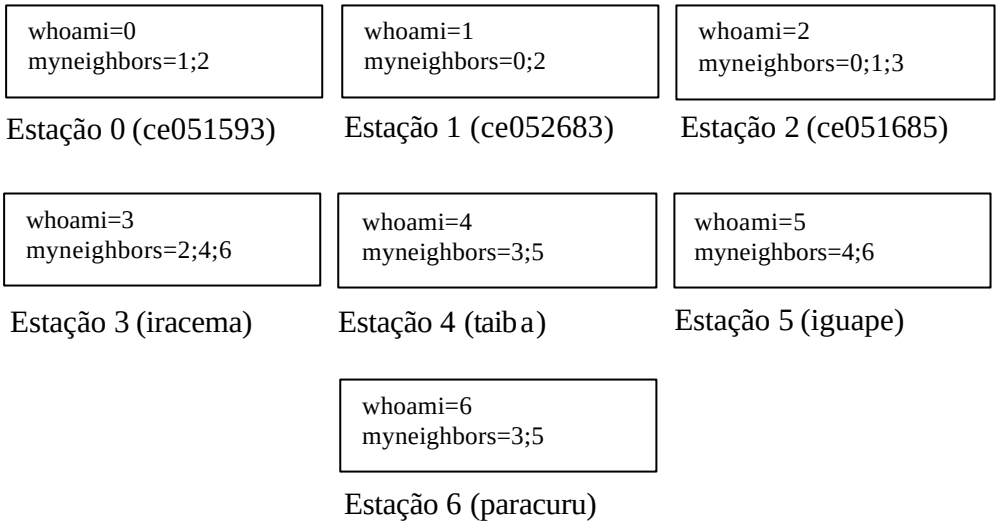


Figura B.2.3 - Conteúdos dos arquivos NODECFG.INF para cada estação da Figura B.2.1

APÊNDICE C

Glossário de Termos

Algoritmo *on-line*: Designação dada ao algoritmo de diagnóstico que opera corretamente mesmo quando falhas e reparações ocorrem durante sua execução.

Algoritmo *off-line*: Designação dada ao algoritmo de diagnóstico que não opera corretamente quando falhas e reparações ocorrem durante sua execução.

Diagnóstico Adaptativo: A identificação das unidades falhas é feita de modo que os testes a serem realizados no futuro são decididos com base nos resultados dos testes atuais. Em outras palavras, a topologia de testes das unidades não é fixa, sendo alterada conforme a situação de falha do sistema, obtida dinamicamente à medida que os testes vão sendo realizados. [HAK 84]

Diagnóstico Automático de Sistema: Nesta dissertação, usamos esta terminologia para nos referirmos ao problema da identificação de unidades falhas em sistemas constituídos de unidades autônomas. Nestes sistemas, assume-se, fundamentalmente, que a presença de qualquer número de unidades falhas não prejudica o funcionamento correto das unidades normais remanescentes. Assume-se ainda que cada unidade é capaz de realizar testes em pelo menos alguma outra. Esta linha de pesquisa é conhecida na literatura técnica de língua inglesa como *System-Level Diagnosis.* A construção "Diagnóstico em Nível de Sistema", no entanto, não é bem aceita na norma oficial da língua portuguesa. Porquanto, a adoção de "*Diagnóstico Automático de Sistema*" nos parece mais apropriada.

Diagnóstico Centralizado: O algoritmo de diagnóstico é executado por uma única unidade, designada como observador central, que é responsável por coletar todos os resultados dos testes (síndrome) e determinar a situação de falha do sistema.

Diagnóstico Distribuído: O algoritmo de diagnóstico é executado de forma independente por todas as unidades normais que compõem o sistema. Não há a figura do observador central presente nos algoritmos de diagnóstico centralizado. [KUH 80,81]

Diagnóstico: Procedimento computacional para a determinação dos estados de funcionamento de todas as unidades de um sistema.

Diagnosticabilidade: Definiremos este termo como sendo o número máximo de unidades que podem falhar num sistema, de modo que ainda seja possível identificá-las precisamente, usando um procedimento de diagnóstico. Este termo é uma adaptação do termo técnico da língua inglesa ***diagnosability***, e não existe na norma oficial corrente da língua portuguesa.

Enlace: Meio físico que conecta duas unidades. Nesta dissertação, os enlaces podem ser ponto a ponto, ligando duas unidades que são ditas *vizinhas físicas*, ou um canal de difusão (ou *broadcast*) de mensagens, conectando duas ou mais unidades que podem, duas a duas, ser convencionadas *vizinhas lógicas* dentro do sistema de diagnóstico.

Estado de Funcionamento: (Ou simplesmente estado) Valor binário associado a uma unidade: 0 (zero) se a unidade está *normal* e 1 (um) se está *falha*. Em alguns algoritmos, como o apresentado nesta dissertação, o estado de funcionamento de uma unidade é determinado pela paridade de um contador de eventos (valor associado à unidade que é incrementado a cada mudança de estado detectada sobre ela). Neste último caso, se o contador de eventos tem um valor *par*, o estado de funcionamento da unidade é considerado *normal*, e, por outro lado, se o contador de eventos tem um valor *ímpar* a unidade é considerada *falha.*

Etapa de Testes: Processamento distribuído no qual todas as unidade realizam todos os testes a que são responsáveis. Para uma dada topologia de testes, uma etapa de testes é a realização de todos os testes que a compõem.

Evento de Falha: Instante em que uma nova falha ocorre no sistema. Transição do estado normal para falho.

Evento de Reparação: Instante em que uma nova reparação ocorre no sistema. Transição do estado falho para normal.

Falha Bizantina: Uma unidade com este tipo de falha continua em operação realizando processamentos incorretos e produzindo saídas espúrias, podendo dar a falsa impressão de que está funcionando normalmente.

Falha *fail-stop* ou *fail-silent*: Uma unidade com este tipo de falha simplesmente cessa sua operação, não sendo capaz de realizar qualquer processamento ou gerar quaisquer saídas espúrias. [SCH 83]

Falha Intermitente: Falha temporária, cujas condições para sua ativação não podem ser reproduzidas ou que ocorra raramente, fazendo com que a unidade apresente um distúrbio em seu comportamento lógico em certos instantes, e apresente um comportamento normal em outros (geralmente a maior parte do tempo). Neste último caso, a unidade pode ser incorretamente considerada normal, quando de fato está falha. [SIE 92]

Falha Permanente: Uma falha que cause a uma unidade exibir *constantemente* um comportamento que é discordante do seu comportamento lógico normal.

Falha Transitória: Falha momentânea que pode afetar o resultado de um teste realizado sobre a unidade, fazendo com que ela possa ser considerada falha, quando na verdade está normal. [SIE 92]

Falha: Uma disfunção no comportamento lógico esperado para uma unidade. As falhas consideradas nesta dissertação são aquelas em que a unidade cessa permanentemente sua operação, não sendo capaz de responder a testes (ainda que incorretamente), nem tampouco gerar relatórios com resultados arbitrários de testes que realize. [SCH 83]

Latência de Diagnóstico: Tempo decorrido entre o instante em que um novo evento de falha ou reparação é detectado até que todas as unidades normais participantes do sistema tenham obtido diagnóstico correto sobre ele.

Nó: O mesmo que unidade de processamento, entretanto, o termo *nó* é mais usual na literatura técnica, para referir-se a unidades (estações) em redes de computadores.

Problema da Caracterização: Dado um certo valor de *t* (o limite superior para o número de unidades falhas), quais são as condições necessárias e suficientes para que um sistema constituído de unidades autônomas seja diagnosticável ?, ou seja, quais são estas condições para que até todas as *t* unidades falhas sejam identificadas indubitavelmente ? [SOM 87]

Problema do Diagnóstico: Dada a topologia de testes de um sistema, tal que até *t* unidades podem falhar e serem *unicamente* identificadas com base em qualquer síndrome, existe um procedimento eficiente (algoritmo) para identificar todas estas unidades falhas ? [SOM 87]

Problema da Diagnosticabilidade: Dada uma coleção de unidades e uma topologia de testes entre elas, qual é o número máximo *t* destas unidades que podem falhar arbitrariamente de modo que ainda sejam possível identificar todas elas *unicamente* com base em qualquer síndrome do sistema ? [SOM 87]

Reparação: Circunstância em que uma unidade previamente falha recupera completamente sua funcionalidade e comportamento normais.

Resultado de um Teste: Um resultado binário associado a um teste. Um valor 0 (zero) significa que o teste teve sucesso; um valor 1 (um) significa que o teste falhou. Na teoria dos grafos, o resultado de um teste é representado através de um peso binário associado ao eixo orientado que representa o teste.

Síndrome: O conjunto formado por todos os resultados dos testes de um sistema *S*. Conceito usando em algoritmos de diagnóstico centralizado.

Sistema: A coleção de um certo número *n* de unidades, formando tipicamente um sistema multiprocessador *S*, conectadas através de uma topologia de interligação. Nesta dissertação, o sistema *S* é a coleção formada por um conjunto de estações interligadas através de uma rede de comunicação de topologia geral. Na teoria dos grafos, o sistema *S* é denotado por um grafo não orientado $G(S) = (V(S),E(S))$, onde $V(S) = \{v_1, v_2, ..., v_n\}$ é o conjunto de todas as unidades e um eixo $(v_x, v_y) \in E(S)$ se, e somente se, há uma interligação entre v_x e v_y.

Situação de Falha: Conjunto formado pelos estados de funcionamento de todas as unidades do sistema. Conceito usando em algoritmos de diagnóstico centralizado.

***t*-Diagnóstico de um Único Passo:** (Também chamado de *diagnóstico sem reparação*) A identificação de todas as unidades falhas (limitadas a *t* unidades no máximo) é possível de ser realizada sem modificar nada no sistema, apenas observando uma síndrome qualquer obtida. [PRE 67]

***t*-Diagnóstico Seqüencial:** (Também chamado de *diagnóstico com reparação*) A identificação de pelo menos uma de no máximo *t* unidades falhas (se existir alguma) no sistema *S* a partir de uma dada síndrome. O diagnóstico é processado em etapas seqüenciais nas quais garante-se que pelo menos uma unidade falha é identificada, se existir. Após a identificação da unidade falha, ela é reparada (ou substituída). Este processo é repetido até que não haja mais nenhuma unidade falha. [PRE 67]

Teste: Procedimento computacional utilizado para determinar o estado de funcionamento de uma unidade.

Topologia de Testes: A coleção de todos os testes em *S* é representada através de um grafo orientado $T(S) = (V(S), R(S))$, onde $V(S) = \{v_1, v_2, ..., v_n\}$ é o conjunto de todas as unidades de *S*, e um eixo $(v_x, v_y) \in R(S)$ se, e somente se, a unidade v_x testa a unidade v_y. Pela definição de sistema *S*, convém observar que se $(v_x, v_y) \in R(S)$, então $(v_x, v_y) \in E(S)$.

Unidade de Processamento: (Ou simplesmente unidade) Um elemento do sistema que é capaz de processar informações e tomar decisões. Nesta dissertação, as unidades são estações de trabalho interligadas através de uma rede de comunicação. Uma unidade é representada na teoria dos grafos como um *vértice v*.