UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CURITIBA

GERÊNCIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL - CPGEI

ANDRÉA WEBER

UM ALGORITMO DE DIAGNÓSTICO DISTRIBUÍDO PARA REDES PARTICIONÁVEIS DE TOPOLOGIA ARBITRÁRIA

TESE DE DOUTORADO

CURITIBA
MAIO 2008

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

---

TESE
apresentada à UTFPR
para obtenção do título de

DOUTOR EM CIÊNCIAS

por

ANDRÉA WEBER

UM ALGORITMO DE DIAGNÓSTICO DISTRIBUÍDO PARA REDES PARTICIONÁVEIS DE TOPOLOGIA ARBITRÁRIA

Banca Examinadora:

Presidente e Orientador:

PROF. DR. ELIAS PROCÓPIO DUARTE JR.	**UFPR**
PROFA. DRA. KEIKO V. ONO FONSECA	**UTFPR**

Examinadores:

PROF. DR. RAIMUNDO J. A . MACÊDO	**UFBA**
PROF. DR. CARLOS BECKER WESTPHALL	**UFSC**
PROF. DRA. SÍLVIA REGINA VERGÍLIO	**UFPR**
PROF. DR. EMÍLIO C. G. WILLE	**UTFPR**

Curitiba, maio de 2008.

Ficha catalográfica elaborada pela Biblioteca da UTFPR – Campus Curitiba

W373a Weber, Andréa
Um algoritmo de diagnóstico distribuído para redes particionáveis de topologia arbitrária / Andréa Weber. Curitiba. UTFPR, 2008
X, 133 f. : il. ; 30 cm

Orientador: Prof. Dr. Elias Procópio. Duarte Jr.
Co-Orientadora: Profª. Drª. Keiko Verônica Ono Fonseca
Tese (Doutorado) – Universidade Tecnológica Federal do Paraná. Programa de Pós Graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2008
Bibliografia: f. 120 – 128

1. Sistemas operacionais distribuídos (Computadores). 2. Algoritmos de computador. I. Duarte Jr., Elias Procópio, orient. II. Fonseca, Keiko Verônica Ono, Co-Orient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD: 004.651

ANDRÉA WEBER

UM ALGORITMO DE DIAGNÓSTICO DISTRIBUÍDO PARA REDES PARTICIONÁVEIS DE TOPOLOGIA ARBITRÁRIA

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial à obtenção do ttulo de "Doutor em Ciências" – Área de Concentração: Telemática.
Orientador: Prof. Dr. Elias Procópio Duarte Jr.
Co-Orientadora: Profa. Dra. Keiko Verônica Ono Fonseca

CURITIBA

2008

AGRADECIMENTOS

Ao meu orientador, Prof. Dr. Elias P. Duarte Jr. pela oriental paciência com que conduziu este trabalho ao longo dos últimos 10 anos.

À minha co-orientadora, Profa. Dra. Keiko V. Ono Fonseca, por ter sido, sobretudo, um apoio amigo em inúmeras ocasiões durante a realização deste trabalho.

À Universidade Tecnológica Federal do Paraná, pela oportunidade de imprimir esta jornada.

Ao Departamento de Informática da Universidade Federal do Paraná, na pessoa de seus chefes de departamento, docentes e funcionários, por terem viabilizado durante 4 anos minha licensa para estudos, sem a qual este trabalho não teria sido possível.

À Universidade Federal do Paraná, pela utilização de seu Laboratório de Processamento de Alto Desempenho, construído com recursos do FINEP.

Aos professores componentes da banca, pelas valiosas contribuições.

À minha família, pelo apoio incondicional.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Adaptive DSD	Algoritmo *Adaptive Distributed System-Level Diagnosis*
DNC	Algoritmo *Distributed Network Connectivity*
DNR	Algoritmo *Distributed Network Reachability*
Hi-ADSD	Algoritmo *Hierachical Adaptive Distributed System-Level Diagnosis*
NBND	Algoritmo *Non-Broadcast Network Diagnosis*
PMC	Algoritmo de Preparata, Metze e Chien
RDZ	Algoritmo de Rangarajan, Dahbura e Ziegler

RESUMO

Este trabalho apresenta um novo algoritmo de diagnóstico distribuído em nível de sistema, *Distributed Network Reachability* (DNR). O algoritmo permite que cada nodo de uma rede particionável de topologia arbitrária determine quais porções da rede estão alcançáveis e inalcançáveis. DNR é o primeiro algoritmo de diagnóstico distribuído que permite a ocorrência de eventos dinâmicos de falha e recuperação de nodos e enlaces, inclusive com partições e *healings* da rede. O estado diagnosticado de um nodo é ou *sem-falha* ou *inatingível*; o estado diagnosticado de um enlace é ou *sem-falha* ou *não-respondendo* ou *inatingível*. O algoritmo consiste de três fases: teste, disseminação e cálculo de alcançabilidade. Durante a fase de testes cada enlace é testado por um de seus nodos adjacentes em intervalos de teste alternados. Após a detecção de um novo evento, o testador inicia a fase de disseminação, na qual a nova informação de diagnóstico é trasmitida para os nodos alcançáveis. A cada vez que um novo evento é detectado ou informado, a terceira fase é executada, na qual um algoritmo de conectividade em grafos é empregado para calcular a alcançabilidade da rede. O algoritmo DNR utiliza o número mínimo de testes por enlace por rodada de testes e tem a menor latência possível de diagnóstico, assegurada pela disseminação paralela de eventos. A correção do algoritmo é provada formalmente. Uma prova de correção no arcabouço *bounded correctness* também foi elaborada, incluíndo *latência delimitada de diagnóstico*, *latência delimitada de inicialização* e *acuidade*. Um simulador do algoritmo foi implementado. Experimentos foram executados em diversas topologias incluindo grafos aleatórios (*k-vertex connected* e *Power-Law*) bem como grafos regulares (*meshes* e hipercubos). Extensivos resultados de simulação de eventos dinâmicos de falha e recuperação em nodos e enlaces são apresentados.

ABSTRACT

This thesis introduces the new *Distributed Network Reachability* (DNR) algorithm, a distributed system-level diagnosis algorithm that allows every node of a partitionable general topology network to determine which portions of the network are reachable and unreachable. DNR is the first distributed diagnosis algorithm that works in the presence of network partitions and healings caused by dynamic fault and repair events. A node is diagnosed as either *working* or *unreachable* and a link is diagnosed either as *working* or *unresponsive* or *unreachable*. The algorithm is formally specified and consists of three phases: test, dissemination, and reachability computation. During the testing phase each link is tested by one of the adjacent nodes at alternating testing intervals. Upon the detection of a new event, the tester starts the dissemination phase, in which the new diagnostic information is received by every reachable node in the network. New events can occur before the dissemination completes. After a new event is detected or informed, a working node runs the third phase, in which a graph connectivity algorithm is employed to compute the network reachability. The algorithm employs the optimal number of tests per link per testing interval and the best possible diagnosis latency, assured by the parallel dissemination of event information. The correctness of the algorithm is proved, including the *bounded diagnostic latency*, *bounded start-up* and *accuracy*. Experimental results obtained from simulation are presented. Simulated topologies include random graphs (k-vertex connected and Power-Law) as well as regular graphs (meshes and hypercubes). Extensive simulation results of dynamic fault and repair events on nodes and links are presented.

CAPÍTULO 1

INTRODUÇÃO

Considere um sistema computacional distribuído composto de várias unidades de processamento autônomas interligadas por canais de comunicação. De forma a permitir que tais sistemas sejam tolerantes a falhas, é fundamental determinar quais de seus componentes estão falhos e quais estão sem-falha. Neste sentido, ferramentas automatizadas de diagnóstico desempenham papel importante na tarefa de prover serviço correto e contínuo. O presente trabalho apresenta um algoritmo de *diagnóstico em nível de sistema* aplicável a redes particionáveis de topologia arbitrária.

A área de diagnóstico em nível de sistema se ocupa de obter informações de diagnóstico sobre as unidades computacionais que compõem um sistema, o qual pode constituir um ambiente computacional distribuído, como é o caso das redes atuais. No decorrer das últimas décadas, vários modelos e algoritmos de diagnóstico foram propostos. Estes algoritmos podem ser classificados em duas famílias: a dos algoritmos de diagnóstico para redes representáveis por grafos completos e a dos algoritmos de diagnóstico para redes de topologia arbitrária.

O primeiro modelo na área de diagnóstico em nível de sistema foi introduzido por [Preparata, Metze e Chien, 1968] e é conhecido como modelo PMC, das iniciais de seus autores. Neste modelo, um sistema é definido como um conjunto de unidades indivisíveis para o propósito de diagnóstico e que possuem capacidade de executar testes e reportar resultados. O modelo pressupõe uma rede totalmente conectada, com a premissa de que os enlaces de comunicação não falham. Os resultados do conjunto de testes executados são submetidos a um observador central, encarregado de fazer o diagnóstico.

Visando obter diagnóstico correto com um número de testes menor do que no modelo PMC, [Hakimi e Nakajima, 1984] propuseram uma abordagem de diagnóstico chamada pelos autores de *adaptativa*. Nesta abordagem, o assinalamento de testes é

adaptado dinamicamente à situação de falhas com base em testes prévios, até encontrar-se uma unidade sem-falha, a partir da qual as demais unidades são diagnosticadas. [Hosseini, Kuhl e Reddy, 1984] propuseram uma abordagem de diagnóstico que não necessita de um observador central. No diagnóstico *distribuído*, cada nodo aceita informações de testes de outros nodos do sistema que tenham sido previamente testados como sem-falha e realiza o diagnóstico do sistema.

O primeiro algoritmo ao mesmo tempo adaptativo e distribuído foi proposto por [Bianchini e Buskens, 1992]. O algoritmo *Adaptive* DSD (*Adaptive Distributed System-Level Diagnosis*) pressupõe uma rede totalmente conectada e o grafo de testes forma um anel, de tal forma que cada nodo é testado por somente um testador por rodada de testes. Sua latência de diagnóstico é de n rodadas de teste, onde n é o número de unidades no sistema. O algoritmo Hi-ADSD, de *Hierachical Adaptive Distributed System-Level Diagnosis*, proposto por [Duarte e Nanya, 1998], também pressupõe uma rede totalmente conectada, mas utiliza uma abordagem hierárquica e adaptativa de diagnóstico, na qual os nodos se organizam em *clusters* de testes. O algoritmo Hi-ADSD apresenta latência de diagnóstico igual a $\log_2^2 n$ rodadas no pior caso.

O trabalho de [Subbiah e Blough, 2004] introduz um modelo formal chamado *Bounded Correctness*, o qual possibilita que um algoritmo de diagnóstico tenha uma prova rigorosa de correção em uma situação dinâmica de ocorrência de eventos. Para ser correto nestas condições, um algoritmo de diagnóstico deve detectar a ocorrência de eventos com latência limitada, deve prover, aos nodos que se recuperam, uma visão dos estados dos demais nodos do sistema com um atraso limitado e não deve detectar eventos espúrios. *Bounded correctness* assume um modelo síncrono de sistema. No mesmo trabalho, os autores apresentam dois algoritmos de diagnóstico baseados em *heartbeats*, para os quais a *bounded correctness* é provada. O algoritmo *HeartbeatComplete* assume uma rede totalmente conectada e considera como falho um nodo do qual não são recebidos *heartbeats* dentro de um intervalo de tempo específico do algoritmo.

Para redes de topologia arbitrária, o primeiro algoritmo de diagnóstico foi proposto por [Bagchi e Hakimi, 1991]. Este algoritmo executa por meio da criação de pacotes

de diagnóstico, os quais são enviados pela rede e podem receber informações de outros. Os pacotes formam uma árvore distribuída de busca em profundidade, através da qual informações de diagnóstico são posteriormente disseminadas. O algoritmo executa *off-line* e pressupõe apenas eventos de falha, que não ocorrem durante sua execução.

O algoritmo *Adapt*, também para redes de topologia arbitrária, foi apresentado por [Stahl, Buskens e Bianchini, 1992a]. Neste algoritmo, um grafo de testes é gerado por intermédio de pacotes transmitidos pelo sistema, daqui em diante também chamado de rede. Posteriormente, a redundância no assinalamento de testes é removida e a informação de diagnóstico é disseminada. O algoritmo prevê eventos de falha e recuperação de nodos, que podem ocorrer durante sua execução.

Os autores [Rangarajan, Dahbura e Ziegler, 1995] propuseram um algoritmo para diagnóstico de redes de topologia arbitária conhecido como RDZ, das iniciais de seus nomes. O algoritmo pressupõe falhas em nodos, os quais não precisam conhecer a topologia da rede, somente seus nodos vizinhos. Os testes são periódicos e em número ótimo, isto é, cada nodo é testado por somente outro nodo do sistema. Nodos falhos deixam de ser testados. Informações de diagnóstico são disseminadas por inundação, com um aprimoramento de forma a minorar as mensagens redundantes. Há uma configuração de falha que o algoritmo não detecta, o que inviabiliza sua utilização.

Também para redes de topologia arbitrária, o algoritmo NBND (*Non-Broadcast Network Diagnosis*) foi proposto por [Duarte, Mansfield, Nanya *et. al.*, 1997]. O algoritmo realiza o diagnóstico de *timeouts* em enlaces e, com base em informação de diagnóstico vinda de outros nodos, calcula a alcançabilidade de nodos na rede. Os enlaces são testados de forma que não somente o nodo testador descobre o estado do nodo testado, como também o nodo testado descobre o estado do nodo testador. Neste sentido um enlace previamente sem-falha, no qual se detecta um *timeout* define um *evento de falha*; um enlace para o qual previamente se detectou um *time-out* e que passa a transmitir uma resposta a um teste define um *evento de recuperação*. Informações de diagnóstico são disseminadas de forma paralela. Mais tarde, uma estratégia de testes baseada em *token* foi proposta por [Siqueira, Fabris e Duarte, 2000], na qual os nodos conectados por

um enlace se alternam nos papéis de testado e testador. No NBND, um novo evento na rede só pode ocorrer após terem sido completamente disseminadas informações sobre eventos anteriores.

O algoritmo DNC (*Distributed Network Connectivity*) [Duarte e Weber, 2003a] foi concebido para redes particionáveis e contempla eventos concorrentes, isto é, durante a disseminação de informações sobre testes, novos eventos podem ocorrer. A disseminação de informação sobre eventos concorrentes é assegurada pela utilização de árvores distribuídas de busca em largura que são concatenadas em diversos pontos da rede até que uma árvore final contendo informações sobre todos os eventos seja formada. Os testes são periódicos e alternados entre os nodos conectados por um enlace. Posteriormente a estratégia de testes do algoritmo foi aperfeiçoada para tratar as situações em que nodos vizinhos que recuperam testam um ao outro simultaneamente [Weber, Duarte e Fonseca, 2006]. Esta nova estratégia de testes garante que somente um dos nodos adote o papel de testador e do próximo intervalo de testes em diante estes papéis sejam alternados. A estratégia também assegura o número mínimo de um testador por enlace por intervalo de testes.

O algoritmo *ForwardHeartbeat* foi introduzido no mesmo trabalho em que o algoritmo *HeartbeatComplete* [Subbiah e Blough, 2004] e utiliza a mesma estratégia de detecção de eventos daquele algoritmo. São definidos eventos em nodos que falham por *crash* ou recuperam. É assumido um sistema síncrono e provas formais no arcabouço de *bounded correctness* são obtidas. O algoritmo é proposto para uma rede de topologia arbitrária, mas assume-se que particionamentos não ocorrem.

O algoritmo *Distributed Network Reachability* (DNR) proposto neste trabalho é o primeiro algoritmo distribuído de diagnóstico em nível de sistema para redes de topologia arbitrária que executa na presença de particionamentos e reconexões da rede. Eventos dinâmicos de falha e recuperação de nodos e enlaces são permitidos. O algoritmo define um novo modelo para representar a alcançabilidade da rede, no qual nodos podem estar nos estados *sem-falha* ou *inatingível* e enlaces podem estar nos estados *sem-falha*, *não-respondendo* ou *inatingível*. O algoritmo consiste de três fases: teste, disseminação e

cálculo de alcançabilidade. Durante a fase de testes, cada enlace é testado por um de seus nodos adjacentes em intervalos de teste alternados, de tal forma que o número de testes executados é o menor possível. A detecção de um novo evento, isto é, um enlace *sem-falha* se torna *não-respondendo* ou vice-versa, desencadeia a fase de disseminação, na qual uma estratégia paralela é utilizada para informar os outros nodos alcançáveis sobre o evento. Novos eventos podem ocorrer e ser detectados antes que a disseminação seja completada, isto é, o algoritmo executa *on-line*. Sempre que um evento é detectado ou informado, a terceira fase é executada, na qual um algoritmo de conectividade em grafos é empregado para computar a alcançabilidade da rede.

Neste trabalho, o algoritmo é especificado e analiticamente provado, incluindo provas formais no arcabouço *bounded correctness*. O algoritmo também é avaliado experimentalmente utilizando técnicas de simulação de eventos discretos.

A principal contribuição deste trabalho é a especificação e prova formal de correção de um algoritmo de diagnóstico em nível de sistema concebido para redes particionáveis de topologia arbitrária, que trata particionamentos e reconexões da rede em situações dinâmicas de ocorrência de eventos. Até a extensão do conhecimento dos autores, este é o primeiro algoritmo de diagnóstico com estas características. O modelo de diagnóstico proposto também pode ser considerado original. Outra contribuição desta tese é a extensão do arcabouço de *bounded correctness* para contemplar a possibilidade de ocorrência de eventos em enlaces. A construção de um simulador permitiu a obtenção de resultados exaustivos de simulação do algoritmo. Também permitiu comparar o algoritmo DNR com o algoritmo *ForwardHeartbeat*, naquelas situações onde esta comparação é possível, tendo em vista que aquele algoritmo não contempla particionamentos na rede.

O restante do texto está organizado como segue. O Capítulo 2 descreve os principais resultados na área de diagnóstico em nível de sistema e outros trabalhos relacionados. O Capítulo seguinte descreve o novo algoritmo de diagnóstico para redes particionáveis de topologia arbitrária, bem como o especifica formalmente e apresenta provas analíticas. As provas formais de correção do algoritmo avaliado segundo o modelo de *bounded correctness* são apresentadas no Capítulo 4. No Capítulo 5 são apresentados extensivos resultados

experimentais da ocorrência de um grande número de eventos dinâmicos em várias topologias de rede, em situações de particionamento e *healing* da rede e em situações em que a rede não particiona. O Capítulo 6 conclui o trabalho. Um Apêndice contém parte das provas formais de correção no arcabouço *bounded correctness*.

CAPÍTULO 2

DIAGNÓSTICO EM NÍVEL DE SISTEMA

Este Capítulo situa a área de *diagnóstico em nível de sistema* no contexto de *tolerância a falhas* e apresenta seus principais resultados. O termo *dependability*, traduzido neste trabalho como *confiança no funcionamento*, é definido como a propriedade que permite aos usuários de um sistema computacional terem uma "confiança justificada no serviço fornecido pelo mesmo" [Avizienis, Laprie, Randell *et. al.*, 2004]. Neste sentido, *serviço* é uma abstração do comportamento do sistema, tal como ele é percebido por seus usuários, automatizados ou não.

Confiança no funcionamento integra atributos tais como *disponibilidade* (*availability*), *confiabilidade* (*reliability*), *segurança* (*safety*), *integridade* (*integrity*) e *manutenibilidade* (*maintainability*) [Avizienis, Laprie, Randell *et. al.*, 2004]. *Disponibilidade* significa estar apto a oferecer serviço correto; *confiabilidade* caracteriza continuidade no fornecimento de serviço correto; *segurança* significa ausência de consequências desastrosas aos usuários e ao meio; *integridade* se refere à capacidade de não sofrer alterações impróprias; e, finalmente, *manutenibilidade* significa habilidade de permitir modificações e reparos.

Neste ponto é oportuno caracterizar as noções de *falha*, *erro* e *defeito*, dos termos em inglês *fault*, *error* e *failure*, respectivamente [Avizienis, Laprie, Randell *et. al.*, 2004]. Um *defeito* ocorre quando o serviço fornecido pelo sistema não está de acordo com a especificação; um *erro* é a parte do sistema que pode levar ao defeito; uma *falha* é a causa provável ou hipotética do erro, e pode estar *ativa* ou *dormente*; erros não detectados são *latentes*. Assim, um defeito é a manifestação, na interface do sistema, da consequência de uma falha ativa.

Como meio de se obter *confiança* é utilizado um grupo de técnicas complementares [Avizienis, Laprie, Randell *et. al.*, 2004]: *prevenção de falhas* (*fault prevention*),

tolerância a falhas (*fault tolerance*), *remoção de falhas* (*fault removal*) e *previsão de falhas* (*fault forecasting*). *Tolerância a falhas*, especificamente, envolve as etapas de *detecção de erros* (*error detection*) e *recuperação do sistema* (*system recovery*). Esta última envolve o *processamento do erro* (*error handling*), significando a remoção do erro do estado computacional; e o *tratamento de falhas* (*fault handling*), com os procedimentos que previnem que falhas localizadas sejam novamente ativadas. O *tratamento de falhas*, por sua vez, envolve, entre outras, a etapa de *diagnóstico de falhas* (*fault diagnosis*), que consiste em "determinar as causas dos erros, em termos de localização e natureza".

Diagnóstico em nível de sistema é uma abordagem que permite que se determine o estado das unidades de um sistema. Este trabalho propõe um novo algoritmo de diagnóstico em nível de sistema aplicável a redes particionáveis de topologia arbitrária. Nas Seções subsequentes, é apresentada uma visão geral da área com alguns dos principais resultados de diagnóstico em nível de sistema. Primeiramente serão apresentados algoritmos concebidos para redes representáveis por grafos completos. Em seguida serão apresentados os resultados para redes de topologia arbitrária. Outras abordagens relacionadas a este trabalho serão apresentadas no final do Capítulo.

2.1 DIAGNÓSTICO DE REDES REPRESENTÁVEIS POR GRAFOS COMPLETOS

Esta Seção apresenta alguns modelos da área de diagnóstico em nível de sistema e a evolução dos algoritmos desta área, concebidos para redes representáveis por grafos de topologia completa.

2.1.1 O Modelo PMC

A área de *diagnóstico em nível de sistema* foi inaugurada pelo trabalho dos autores [Preparata, Metze e Chien, 1968]. Naquele trabalho, um *sistema* é caracterizado como um conjunto de unidades, onde uma *unidade* é uma porção do sistema não decomponível para propósitos de diagnóstico. As unidades possuem capacidade de testar outras unidades e o fazem por meio de um *assinalamento de testes*. Assume-se que os enlaces de comunicação nunca falham, qualquer unidade pode testar outra unidade, ou seja,

o sistema é representável por um grafo completo e nenhuma unidade testa a si mesma.

As unidades podem estar nos estados *falho* ou *sem-falha* e toda unidade sem-falha é capaz de realizar testes com perfeição, enquanto que uma unidade falha produz diagnóstico arbitrário, quaisquer que sejam as condições da unidade testada. O modelo prevê um tipo arbitrário de falha, desde que passível de detecção pelas unidades do sistema. Um teste envolve a aplicação de um conjunto de estímulos e a coleta das respostas correspondentes. O conjunto de resultados de teste de todas as unidades do sistema é chamado de *síndrome*. Assume-se que a síndrome do sistema é submetida a um observador central, responsável por *decodificá-la* e obter o diagnóstico.

Além disso, o trabalho de [Preparata, Metze e Chien, 1968] trata um sistema em termos de sua *diagnosability*, que traduzimos como *diagnosticabilidade*. Um sistema composto por n unidades é definido como *t-diagnosticável* se todas as suas unidades falhas podem ser identificadas desde que o número de unidades falhas não exceda um parâmetro t definido pela estrutura do grafo de testes. Prova-se que em um sistema *t-diagnosticável*, n deve ser maior ou igual a $2t+1$ e cada unidade deve ser testada por pelo menos t outras unidades. desde que duas unidades não se testem mutuamente [Hakimi e Amin, 1974]. Um sistema *t-diagnosticável* é definido como *ótimo* se $n = 2t + 1$ e se cada unidade é testada por exatamente t unidades.

2.1.2 Diagnóstico Adaptativo

O modelo PMC [Preparata, Metze e Chien, 1968] assume que o teste feito por uma unidade sem-falha é sempre correto, enquanto que o teste feito por uma unidade falha pode ou não ser correto, independente do estado da unidade testada. Esta asserção é conhecida como modelo de invalidação simétrico de testes (*symmetric invalidation*), enquanto que outro modelo, o de invalidação assimétrica de testes (*asymmetric invalidation*), proposto por [Barsi, Grandoni e Maestrini, 1976], caracteriza que uma unidade falha não pode avaliar outra unidade falha como sem-falha.

No diagnóstico clássico, tendo-se escolhido um conjunto *fixo* de testes e respeitando-se o limite de diagnosticabilidade, obtém-se a síndrome do sistema e identifica-se as unida-

des falhas. [Hakimi e Nakajima, 1984] sugerem que, mantendo-se o limite de unidades falhas mencionado acima, o conjunto de testes seja escolhido de forma *adaptativa* com base nos resultados já obtidos, até encontrar no mínimo uma unidade sem-falha. A partir desta unidade se fazem testes para diagnosticar as demais unidades.

Seguindo este raciocínio, os autores apresentam um algoritmo para o modelo de invalidação simétrica de testes, que identifica a primeira unidade sem-falha com no máximo $2t - 1$ testes. Com mais $(n - 1)$ testes pode-se avaliar as demais unidades. Isto implica, de imediato, em no máximo $n + 2t - 2$ testes para identificar todas as unidades falhas.

Os autores também apresentam algoritmos adaptativos para o modelo de invalidação assimétrica de testes, mas apontam como principal limitação de sua teoria a premissa de que cada unidade do sistema tenha habilidade para testar qualquer outra unidade do mesmo.

2.1.3 Diagnóstico Distribuído

Paralelamente à abordagem sugerida por [Hakimi e Nakajima, 1984], e devido aos inconvenientes de se ter um observador central que faça o diagnóstico, surge uma abordagem *distribuída* para o diagnóstico. [Hosseini, Kuhl e Reddy, 1984] apresentaram o algoritmo New-SELF. Neste algoritmo, a informação utilizada por um nodo para fazer o diagnóstico é proveniente de testes feitos por aquele nodo em outras unidades do sistema ou por informações de testes realizados por outras unidades e enviadas às demais. Como o envio de mensagens de diagnóstico pelo sistema, daqui em diante também chamado de rede, é também sujeito a falhas, isto é tratado pelo algoritmo assumindo que um nodo que detecta um enlace ou nodo como falho deixa de aceitar mensagens de diagnóstico através daquele recurso.

Neste algoritmo, o modelo de falhas assume que nodos podem falhar de modos outros além da falha do tipo *crash*, desde que passíveis de detecção pelos demais nodos. Enlaces também podem falhar e assume-se o uso de códigos detectores de erro que tornariam evidentes as falhas em enlace. Assume-se, ainda, *ambiguidade de falhas* ao se detectar dificuldade de comunicação com nodos vizinhos: nestes casos, não é possível sa-

ber se é o nodo vizinho ou o enlace de comunicação que está falho. Entretanto, enlaces de comunicação falhos entre um par de nodos sem-falha são corretamente detectados como tal.

Não somente falhas são diagnosticáveis pelo algoritmo, mas também a recuperação ou retorno de unidades falhas ao sistema são permitidas, e a introdução de nodos ou enlaces completamente novos ao sistema é possível.

O assinalamento de testes é fixo, e determinado por um grafo de testes, direcionado, que representa o sistema e os nodos que testam cada nodo; cada nodo testa seus vizinhos no grafo de testes e envia mensagens de teste a todos os seus testadores. Assim, as informações de diagnóstico seguem o caminho inverso do caminho seguido pelos testes, garantindo sua confiabilidade.

Um nodo somente aceita informações de diagnóstico de outros nodos que ele testa e diagnostica como sem-falha. O algoritmo é executado *on-line*, no sentido em que nodos podem falhar e serem reparados a qualquer tempo, com a restrição de que um nodo não pode falhar e recuperar de forma não detectável durante o intervalo entre dois testes consecutivos por outro nodo.

Quando ocorre o diagnóstico de um enlace falho por determinado nodo, o outro nodo conectado pelo mesmo enlace é considerado como suspeito de falha até que receba a informação do diagnóstico ambíguo e envie, por sua vez, a informação de que é o enlace que se encontra falho. Isso evidencia que, embora enlaces possam falhar, as falhas não podem desconectar a rede.

À semelhança do algoritmo de [Hakimi e Nakajima, 1984], o algoritmo New-SELF converge para diagnóstico correto desde que menos do que t nodos estejam sem-falha, onde $t \leq (n-1)/2$ e n é o número de nodos no sistema. Na eventualidade de um número maior do que t, possivelmente todas as unidades do sistema, se tornem falhas, o algoritmo recobra o diagnóstico correto tão logo a medida de diagnosticabilidade, t, seja restaurada. Ao adicionar novos nodos ou enlaces ao sistema, deve-se fazê-lo de forma a manter o limite t de diagnosticabilidade.

Idealmente, um nodo deve ser testado por apenas um nodo sem-falha para garantir

diagnóstico correto; assim, a proposta dos autores do New-SELF, embora inovadora em termos da distribuição dos mecanismos de detecção de falhas, peca pela redundância no assinalamento de testes, o que leva, por sua vez, à redundância na transmissão de informações sobre testes pela rede.

2.1.4 O Algoritmo *Adaptive* DSD

O algoritmo *Adaptive* DSD foi proposto e implementado por [Bianchini e Buskens, 1992]. É um algoritmo de diagnóstico *distribuído*, onde cada nodo faz o diagnóstico da rede, e *adaptativo*, no qual a configuração de testes varia conforme a configuração de falhas do sistema em dado momento.

O algoritmo prevê falhas e recuperações de nodos, mas o modelo não prevê falhas de enlaces e a topologia deve ser totalmente conectada. Como o algoritmo é adaptativo, o número permitido de nodos falhos é limitado a $n - 1$, isto é, havendo apenas um nodo sem-falha, este é capaz de determinar o estado de todos os demais nodos do sistema e corretamente fazer o diagnóstico.

Cada nodo testa apenas um conjunto limitado de nodos e recebe resultados de testes executados por outros nodos. Resultados obtidos de outros nodos são validados não aceitando informações de testes reportadas por nodos testados como falhos (técnica de *report validation*).

2.1.4.1 Descrição do Algoritmo

O algoritmo consiste em fazer testes progressivos até encontrar um nodo sem-falha e atualizar informações locais sobre o estado da rede com informações recebidas daquele nodo. A busca de um nodo sem-falha é feita de maneira sequencial em termos do identificador de cada nodo e a adição é feita em *módulo n*, onde n é o número de nodos no sistema. O grafo de testes forma, então, um ciclo, conforme pode ser visto na Figura 2.1. Cada nodo executa o algoritmo em intervalos de teste e a informação de diagnóstico é enviada através da rede no sentido inverso àquele dos testes no grafo de testes. Na Figura 2.1, o nodo 0 testa o nodo 1 e, o tendo encontrado falho, testa em seguida o nodo 2, de quem

recebe informação de diagnóstico. À semelhança disto, o nodo 3 testa os nodos 4 e nodo 5 antes de testar o nodo 6 como sem falha. Os demais nodos testam como sem-falha o próximo nodo na sequência, respectivamente.

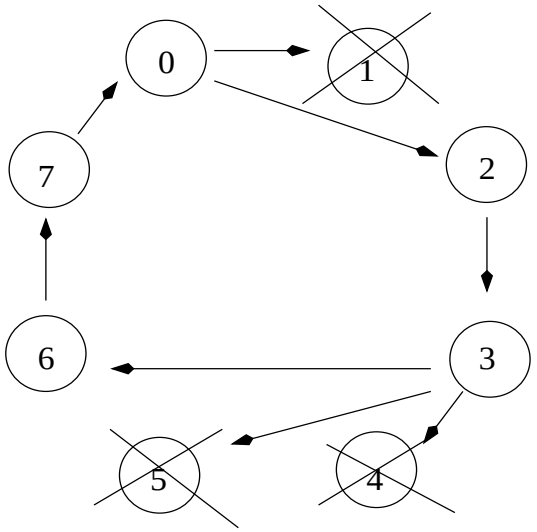

Figura 2.1: Grafo de testes do algoritmo *Adaptive* DSD para um sistema com 8 nodos.

Um vetor denominado TESTED_UP[n] é mantido localmente em cada nodo e TESTED_UP[i]=j significa que o nodo i testou o nodo j como *sem-falha*. Um valor arbitrário é mantido nas entradas correspondentes aos nodos falhos, cujo resultado de testes não é confiável. A Tabela 2.1, reproduzida de [Bianchini e Buskens, 1992], mostra o vetor TESTED_UP[n] mantido, por exemplo, no nodo 2, para a configuração de falhas mostrada na Figura 2.1.

Uma vez obtida informação sobre os testes, o diagnóstico é realizado localmente executando um algoritmo, *Diagnose*, o qual utiliza como base o vetor TESTED_UP[n] atualizado. O diagnóstico do sistema é armazenado no array STATE[n]: STATE[i] contém, para cada nodo i, seu estado como falho ou sem-falha.

O algoritmo *Diagnose* utiliza as entradas sem-falha de TESTED_UP[n]. STATE[n] é inicializado como falho e um ponteiro é inicializado com o identificador do nodo executando o algoritmo. O algoritmo atravessa o "caminho" de nodos sem-falha armazenado em TESTED_UP[n] assinalando cada nodo "visitado" como sem-falha no vetor STATE[n]. O algoritmo é executado até retornar ao nodo onde iniciou; neste ponto, se tem o diagnóstico

do sistema.

Tabela 2.1: Array TESTED_UP[n].

TESTED_UP[0] = 2
TESTED_UP[1] = x
TESTED_UP[2] = 3
TESTED_UP[3] = 6
TESTED_UP[4] = x
TESTED_UP[5] = x
TESTED_UP[6] = 7
TESTED_UP[7] = 0

2.1.4.2 Desempenho do Algoritmo

Os autores provam que os testes executados pelos nodos sem-falha formam um ciclo após o algoritmo *Adaptive* DSD ter sido executado ao menos uma vez em cada nodo, fazendo com que as entradas de TESTED_UP[n] sejam consistentes em todos os nodos sem-falha.

Para isso, define-se uma *rodada de testes* como o período de tempo no qual *Adaptive* DSD executa pelo menos uma vez em cada nodo sem-falha do sistema. Após uma rodada de testes, existe um caminho de cada nodo sem-falha até qualquer outro nodo sem-falha no sistema. Este caminho forma um ciclo. As entradas de TESTED_UP[n] são as mesmas após um número limitado de rodadas de teste depois da ocorrência de um evento. Como o maior caminho possível pelo ciclo contém n nodos, então em até n rodadas de teste todos os nodos sem-falha atualizam TESTED_UP[n].

Como o algoritmo *Diagnose* utiliza somente as entradas sem-falha de TESTED_UP[n] para o diagnóstico, este resulta correto em todos os nodos sem-falha.

O algoritmo *Adaptive* DSD, portanto, é ótimo em termos do número total de testes necessários, pois cada nodo é testado por no máximo um nodo. A latência de diagnóstico, entretanto, é de nT_r, onde T_r é o tempo de um intervalo de teste e n é o número de nodos no sistema.

2.1.5 O Algoritmo Hi-ADSD

O algoritmo Hi-ADSD, ou *Hierarchical Adaptive Distributed System-level Diagnosis*, foi proposto por [Duarte e Nanya, 1998]. Como o próprio nome diz, o algoritmo é adaptativo em termos da estratégia de testes e o diagnóstico é distribuído. O princípio hierárquico do algoritmo consiste em agrupar os nodos em *clusters* de teste. Ao testar um dos nodos de determinado *cluster* obtêm-se informação de diagnóstico sobre todos os nodos daquele cluster.

O algoritmo assume um sistema com n nodos, representável por um grafo de topologia completa. Os nodos podem estar nos estados *falho* e *sem-falha*. Não há limite para o número de nodos falhos, mas falhas de enlace não são consideradas. O algoritmo funciona em rodadas de teste, sendo a latência de diagnóstico no pior caso igual a $\log_2^2 n$ rodadas em um sistema com n nodos.

2.1.5.1 Definindo os *Clusters* de Teste

Os *clusters* são grupos de nodos para os propósitos de teste, conforme mostra a Figura 2.2, e os testes são executados periodicamente. A cada intervalo de teste, um nodo testa sequencialmente os nodos de determinado *cluster* até encontrar um nodo *sem-falha*, a partir do qual obtém informações de diagnóstico sobre os demais nodos daquele *cluster*.

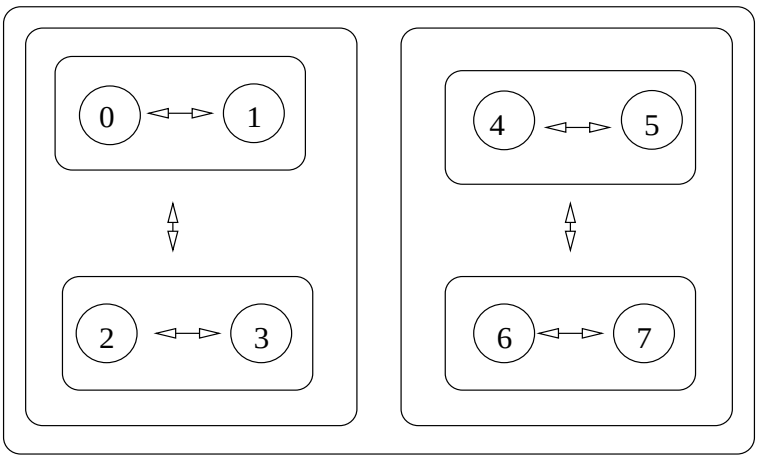

Figura 2.2: Nodos agrupados em *clusters*.

Os *clusters* testados são de tamanhos progressivamente maiores a cada intervalo

de testes. Os tamanhos são expressos em potências de 2, desde o *cluster* com um nodo até o *cluster* de tamanho máximo igual a $n/2$ nodos, sendo o sistema como um todo um *cluster* com n nodos. Após o teste do *cluster* com $n/2$ nodos torna-se a testar um *cluster* de tamanho 1 e assim sucessivamente.

Se todos os nodos de determinado *cluster* estiverem falhos, o nodo passa a testar o próximo *cluster* no mesmo intervalo de teste, até encontrar um nodo sem-falha ou testar todos os nodos do sistema como falhos.

A sequência de *clusters* testados pelo nodo com identificador 0 num sistema com 8 nodos é mostrada na Figura 2.3.

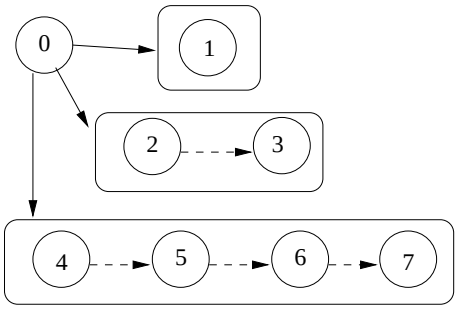

Figura 2.3: Os testes adaptativos nos *clusters*.

O identificador do primeiro nodo a ser testado em cada *cluster* e a sequência dos nodos a serem testados naquele *cluster* variam conforme o identificador do nodo testador. A função que determina estas sequências de nodos é denominada $C_{i,s}$, sendo i o identificador do nodo testador e 2^{s-1} o tamanho do *cluster* a ser testado. A Tabela 2.2 mostra as sequências de nodos para um sistema com 8 nodos [Duarte e Nanya, 1998].

Tabela 2.2: $C_{i,s}$ para um sistema com 8 nodos.

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

Por exemplo, do ponto de vista do nodo 0, o primeiro nodo a ser testado é o nodo 1, no *cluster* de tamanho 1; no *cluster* de tamanho 2, o primeiro nodo a ser testado é o nodo

2 e, em seguida, o nodo 3; no *cluster* de tamanho 4, o primeiro nodo é o de identificador 4, em seguida o nodo 5, depois o nodo 6 e por último o nodo 7, tornando ao *cluster* de tamanho 1 no próximo intervalo de testes. Já do ponto de vista do nodo 1, o primeiro nodo a ser testado é o nodo de identificador 0, no *cluster* de tamanho 1; depois os nodos 3 e nodo 2, nesta ordem, no *cluster* de tamanho 2; e os nodos 5, nodo 6, nodo 7 e nodo 4, nesta ordem, no *cluster* de tamanho 4; e, assim, de maneira análoga para os demais nodos, de acordo com $C_{i,s}$.

Embora os testes ocorram em intervalos de teste, o sistema como um todo é assíncrono, significando que diferentes nodos no sistema podem estar testando *clusters* de diferentes tamanhos em determinado momento.

2.1.5.2 Desempenho do Algoritmo

A latência de diagnóstico do Hi-ADSD é definida em termos de *rodadas de teste*, que são definidas como o "período de tempo no qual todo nodo sem-falha no sistema testou outro nodo sem-falha" e obteve informações de diagnóstico deste último. Assim, a latência de diagnóstico é o número de rodadas de teste necessárias para que todos os nodos sem-falha no sistema obtenham diagnóstico correto e está limitada a $\log_2^2 n$ para um sistema com n nodos. Embora este número seja para o pior caso, resultados experimentais mostram que o diagnóstico é obtido em número menor de rodadas de teste, sendo, na média, da ordem de $O(\log n)$.

A Tabela 2.3 [Duarte e Nanya, 1998] mostra resultados comparativos de latência entre os algoritmos *Adaptive* DSD e Hi-ADSD, para sistemas com n nodos, lembrando que a latência de diagnóstico do algoritmo *Adaptive* DSD é igual a n rodadas de teste.

2.1.6 Diagnóstico de Eventos Dinâmicos

Em seu trabalho, [Subbiah e Blough, 2004] propõem um modelo formal intitulado *Bounded Correctness*, o qual define parâmetros para correção de algoritmos de diagnóstico distribuído em redes com eventos dinâmicos. O modelo se caracteriza por três propriedades, *bounded diagnostic latency*, *bounded start-up* e *accuracy*, as quais serão descritas a

Tabela 2.3: Latências de diagnóstico: ADSD × Hi-ADSD.

n	Hi-ADSD	*Adaptive* DSD
4	4	4
8	9	8
16	16	16
32	25	32
64	36	64
128	49	128
256	64	256
512	81	512
1024	100	1024

seguir.

O trabalho apresenta também dois algoritmos, para os quais *bounded correctness* é provada. Um dos algoritmos é concebido para redes que usam comunicação por *unicast* (*e.g.*, redes ponto a ponto) e outro para redes que usam comunicação por *multicast* (*e.g.*, redes totalmente conectadas). Para ambos os casos, assume-se que o atraso na comunicação é limitado, ou seja, que o sistema é síncrono.

Assume-se que a rede entrega mensagens de maneira confiável e permanece conectada o tempo todo, de forma que o sistema não sofra particionamento. Mensagens incompletas são percebidas como falhas de omissão nos nodos receptores. Nodos sofrem falhas do tipo *crash* e podem estar nos estados *failed* e *working*.

Para as redes totalmente conectadas, nenhuma restrição é imposta para o número de nodos que podem estar no estado falho em determinado instante de tempo, nem para o número de eventos que podem ocorrer simultanemente. Para redes não totalmente conectadas, um número de nodos menor que o da conectividade de vértices da rede pode estar no estado falho em qualquer instante de tempo.

O modelo impõe uma restrição formalizada através da definição de *state holding time*, como o tempo mínimo que cada nodo deve permanecer em determinado estado de forma que a transição para aquele estado seja detectada.

Os algoritmos utilizam mensagens de *heartbeat*, e cada nodo sem-falha periodicamente inicia uma rodada dessas mensagens para os demais nodos de forma a indicar que está sem-falha; π é definido como o *período de heartbeat*.

Um nodo não espera mais do que π, em tempo local, para enviar *heartbeats* após uma recuperação e o intervalo de tempo definido como *recovery wait time*, $W \leq \pi$, é o tempo local pelo qual o algoritmo espera após entrar no estado de *working* antes de iniciar uma rodada de transmissões de *heartbeats*. O *recovery wait time* é útil para permitir que o tempo mínimo que determinado nodo deve permanecer no estado falho seja diminuto, mesmo assim permitindo a detecção daquele estado.

São definidos, ainda, os tempos mínimo e máximo do atraso da transmissão de mensagens num canal de comunicação entre dois nodos vizinhos sem-falha, como Δ_{send_min} e Δ_{send_max}, respectivamente. O tempo entre um nodo iniciando uma comunicação e o último *bit* da mensagem sendo colocado na rede é definido como Δ_{send_init}. A oscilação máxima que um relógio pode sofrer no sistema é definida como $\rho << 1$.

Em um sistema sofrendo eventos dinâmicos e com um atraso não nulo na comunicação entre os dois nodos, as visões que estes têm do sistema são, também, atrasadas. Assim, o estado armazenado por um nodo sem-falha, sobre outro nodo no sistema é dito *T-válido* se este último nodo esteve naquele estado em algum instante durante o intervalo de tempo $[t - T, t]$.

Com isso, as propriedades mínimas que um algoritmo deve garantir são: detectar cada novo evento no sistema tão rápido quanto possível; permitir que os nodos obtenham uma visão dos estados dos outros nodos limitadamente atrasada ao recuperar; e não detectar eventos espúrios. *Bounded correctness* é definida respectivamente em termos de *bounded diagnostic latency*, *bounded start-up* e *accuracy*.

A propriedade de *bounded diagnostic latency* assegura que, para qualquer evento ocorrido no tempo t, qualquer nodo que se mantenha sem-falha continuamente durante $[t, t + L]$ obtém informação sobre o mesmo no máximo ao tempo $t + L$, onde L é a latência de diagnóstico do algoritmo.

Bounded start-up define que um nodo que se recupera ao tempo t e permanece sem-falha continuamente durante $[t, t + S]$, no máximo ao tempo $t + S$ obtêm estados *L-válidos* de todos os outros nodos no sistema, sendo $S \geq L$ definido como o tempo de *start-up* do algoritmo.

Accuracy estabelece que qualquer transição de estado mantida por um nodo sobre outro nodo corresponda a um evento real neste último e nenhum evento singular em um nodo seja interpretado como múltiplas transições de estado por outro.

2.1.6.1 O Algoritmo *HeartbeatComplete*

Em seu trabalho, [Subbiah e Blough, 2004] propõem um modelo formal intitulado *Bounded Correctness*, o qual define parâmetros para correção de algoritmos de diagnóstico distribuído em redes com eventos dinâmicos. O trabalho apresenta também um algoritmo concebido para redes que usam comunicação por *unicast* (*e.g.*, redes ponto a ponto).

Assume-se que o atraso na comunicação é limitado, ou seja, que o sistema é síncrono. Assume-se também que a rede entrega mensagens de maneira confiável e permanece conectada o tempo todo. Mensagens incompletas são percebidas como falhas de omissão nos nodos receptores. Os nodos sofrem falhas do tipo *crash* e podem estar nos estados *failed* e *working*.

Sendo a rede totalmente conectada, nenhuma restrição é imposta para o número de nodos que podem estar no estado falho em determinado instante de tempo, nem para o número de eventos que podem ocorrer simultanemente. A única restrição imposta pelo modelo é formalizada através da definição do *state holding time*, ou *tempo de retenção de estado*, como o tempo mínimo que cada nodo deve permanecer em determinado estado de forma que a transição para aquele estado seja detectada.

O algoritmo *HeartbeatComplete*, introduzido no trabalho, baseia-se no fato de que, se mensagens de *heartbeat* são enviadas periodicamente utilizando mecanismos de *broadcast*, um nodo pode ser considerado falho se nenhum *heartbeat* é recebido dele em um intervalo de tempo específico do algoritmo denominado $\Delta_{heartbeat}$. Se isto ocorrer, os nodos não falhos atualizam o estado do nodo cujos *heartbeats* não chegaram como *failed*. Quando um *heartbeat* de um nodo chega aos demais, seu estado nos demais nodos é assinalado como *working* e o temporizador de espera do próximo *heartbeat* daquele nodo é novamente iniciado.

Os autores provam que o algoritmo *HeartbeatComplete* atinge *bounded correctness*

e caracterizam suas latências de diagnóstico e de inicialização e tempos de retenção de estado.

2.2 ALGORITMOS DE DIAGNÓSTICO PARA REDES DE TOPOLOGIA ARBITRÁRIA

Nesta Seção é mostrada a evolução dos algoritmos de diagnóstico em nível de sistema para redes de topologia arbitrária. Em uma rede de topologia arbitrária não existe necessariamente um enlace de comunicação entre quaisquer pares de nodos. Os algoritmos são expostos em ordem cronológica.

2.2.1 O Algoritmo de Bagchi e Hakimi

Um algoritmo totalmente distribuído foi proposto por [Bagchi e Hakimi, 1991], concebido para um sistema constituído de n processadores idênticos, cada qual com identificação única e conectados por uma estrutura de comunicação arbitrariamente estruturada. Inicialmente cada unidade conhece somente sua identificação e a identificação de seus vizinhos, sem que nenhuma unidade tenha conhecimento global da topologia.

As falhas são permanentes e ocorrem antes do início do algoritmo, de forma que falhas intermitentes não são consideradas. Falhas podem ser do tipo *crash* ou não, desde que diagnosticáveis. Um processador sem-falha sempre faz diagnóstico correto e enlaces de comunicação não podem falhar. Considera-se que a transmissão nos enlaces é feita somente em uma direção de cada vez, e de acordo com uma disciplina FIFO (*first-in, first-out*).

O algoritmo baseia-se na transmissão de pacotes por unidades sem-falha. Assumindo que o sistema forma um grafo conexo com as unidades sem-falha, ao final da execução do algoritmo toda unidade sem-falha sabe o estado de cada unidade do sistema. Alternativamente, em cada componente conexo que o algoritmo for executado, todas as unidades sem-falha saberão os estados de todas as unidades sem-falha daquele componente conexo e também o estado de seus vizinhos falhos.

2.2.1.1 Descrição do Algoritmo

Devido à dificuldade de arbitrar uma unidade sem-falha para iniciar o algoritmo, assume-se que as unidades "acordam" espontânea e aleatoriamente e iniciam sua execução. O comportamento de unidades falhas ao "acordar" é imprevisível. O algoritmo é iniciado com a criação de um pacote com informações de diagnóstico e a consequente formação de uma árvore distribuída de busca em profundidade para disseminá-lo, constituída somente de nodos sem-falha. A busca dos nodos para formação da árvore inicia-se pelos nodos vizinhos, no estágio chamado de *coleta*. No próximo estágio, de *broadcast*, um único pacote percorre a árvore recém-formada levando informações de diagnóstico para todos os nodos da rede. Mesmo que dois ou mais nodos iniciem a formação concomitante de árvores de busca, as diversas árvores serão "concatenadas" utilizando um critério baseado no tamanho das mesmas.

Estendendo uma Árvore O processo de buscar nodos sem-falha, enviar-lhes um pacote e esperar pelo retorno do mesmo, após o processo ter sido repetido pelos nodos que o receberam, é denominado pelos autores de *estender uma árvore*.

O pacote para criação de uma árvore contém os seguintes campos: $PKT.id$, que identifica o pacote e o nodo que o criou como raiz da árvore; $PKT.size$, que contém o número de árvores concatenadas para formar a árvore atual (inicialmente 1); $PKT.visited$, que contém uma lista das unidades diagnosticadas ao longo da formação da árvore, juntamente com seu estado; e, finalmente, $PKT.alldone$, com dupla função conforme a fase do algoritmo que estiver sendo executada, conforme será visto adiante.

Os pacotes e suas respectivas árvores são concatenados sempre que se encontram no decorrer da execução do algortimo. Um pacote criado pela uniidade i é designado como $PKT(i)$ e sua árvore como T_i. Os tamanhos ($PKT.size$) e identificadores ($PKT.id$) dos pacotes são comparados da seguinte forma: $PKT(i) > PKT(j)$ se $\log PKT(i).size > \log PKT(j).size$ ou se $\log PKT(i).size = \log PKT(j).size$ e $i > j$.

Supondo que $PKT(i) > PKT(j)$ e os pacotes PKT(i) e PKT(j) estejam sendo concatenados, os campos do novo pacote ficam como: $PKT(i+j).id = PKT(i).id$, ou

seja, prevalece a árvore com identificador maior conforme critério de comparação acima; $PKT(i+j).size = PKT(i).size + PKT(j).size$; $PKT(i+j).visited = PKT(i).visited \cup PKT(j).visited$. O campo PKT.*alldone* não é afetado pela concatenação de pacotes.

Cada unidade que participa do processo de estender a árvore mantém localmente as seguintes listas de nodos: *latest child*, *other children* e *father*. Supondo que esteja sendo criada a árvore T_i e que a unidade i fez o diagnóstico de uma unidade vizinha, e assumindo que ambas as unidades estejam sem-falha, isso "acorda" a unidade vizinha e ambas se diagnosticam mutuamente. Neste caso, a unidade vizinha é incluida em $PKT(i).visited$ e em *latest child* na unidade que a diagnosticou, e recebe seu pacote. Em seguida, a unidade diagnosticada assinala a unidade que enviou o pacote como seu pai, *father*, na árvore e repete o processo acima.

Se, no entanto, a unidade diagnosticada está falha, o pacote não lhe é enviado e a unidade é incluída em $PKT(i).visited$ como unidade falha. Neste caso, a unidade testadora tenta encontrar outra unidade para quem enviar o pacote. Se, por outro lado, uma unidade falha cria um pacote, ela também é diagnosticada ao enviá-lo, e, sendo falha, tem seus pacotes ignorados pelas unidades que os recebem.

Durante este estágio, de *coleta*, em algum ponto será alcançada uma unidade, a qual terá todos os seus vizinhos incluídos em $PKT(i).visited$. Neste ponto, a unidade retorna o pacote a seu pai com o campo $PKT(i).alldone$ assinalado como 1, indicando que o trabalho está concluído naquela sub-árvore. O nodo pai desta unidade a retira do seu conjunto *latest child*, a inclui no conjunto *other children*, assinala $PKT(i).alldone = 0$ e tenta achar um novo vizinho para incluir na árvore. Desta forma, quando o pacote retornar para a raiz da árvore e todos os vizinhos da raiz estiverem na lista $PKT(i).visited$, o pacote conterá os estados de todos os nodos da rede. No estágio seguinte, de *broadcast*, $PKT(i).alldone = 1$ assinala que o pacote contém os estados de todas as unidades no sistema.

Concatenando Árvores Supondo que haja dois pacotes percorrendo a rede e criando árvores, estas são concatenadas em uma única árvore da seguinte forma: uma

variável local a cada nodo, *current_pkt*, armazena uma cópia do pacote que identifica a árvore da qual o nodo faz parte. Supondo duas árvores concorrentes, T_i e T_j, e duas unidades do sistema, u pertencente a T_i e v pertencente a T_j, se um pacote *PKT(i)* chega de u para v e se $PKT(i) < PKT(j)$, então $PKT(i)$ é deixando à espera em u. Quando $PKT(j)$ volta a v, esta unidade vai *capturar* $PKT(i)$: ambos os pacotes serão concatenados e T_i se tornará uma sub-árvore de T_j.

Se, por outro lado, $PKT(i) > PKT(j)$, $PKT(i)$ é enviado pelo mesmo caminho que $PKT(j)$, tornando T_j uma sub-árvore de T_i. Devido à disciplina FIFO nos enlaces e porque dois pacotes não podem cruzar um enlace simultaneamente, é certo que ambos deverão se encontrar em alguma unidade para serem combinados em um único pacote.

Os autores demonstram que não é possível ocorrer um *deadlock* quando um dos nodos é colocado à espera durante o processo de combinação de árvores. Os autores também demonstram que os pacotes não andam em ciclos pela rede durante o processo de formação de árvores.

Se p unidades sem-falha iniciarem o algoritmo simultaneamente e houver t unidades falhas, o algoritmo requer no máximo $n-1+p(t+1)$ operações de diagnóstico e no máximo $3n \log p + O(n + pt)$ mensagens transmitidas por unidades sem-falha.

2.2.2 O Algoritmo *Adapt*

Logo após a publicação do algoritmo *Adaptive* DSD [Bianchini e Buskens, 1992], para redes de topologia completa, [Stahl, Buskens e Bianchini, 1992a] publicaram o algoritmo *Adapt*, para redes de topologia arbitrária. A proposta prevê execução *on line*, ao contrário do algoritmo de [Bagchi e Hakimi, 1991], que apenas pode ser executado *off line*. O modelo de falhas adotado foi o PMC [Preparata, Metze e Chien, 1968] e há testes periódicos. Assume-se que um nodo não pode falhar e recuperar de maneira não detectável no intervalo entre dois testes consecutivos. São assumidos também códigos de correção de erro para assegurar que mensagens geradas por nodos falhos sejam descartadas.

O algoritmo constrói um *assinalamento de testes*, isto é, um grafo de testes onde

os vértices são os nodos do sistema e as arestas representam um teste executado por determinado nodo em outro. O assinalamento de testes ocorre de maneira distribuída e adaptativa e, com ele, o diagnóstico é feito. Se o grafo ficar desconectado, o algoritmo opera corretamente em cada um dos componentes conectados.

2.2.2.1 Descrição do Algoritmo

O algoritmo opera gerando um assinalamento de testes em duas fases: *Search* e *Destroy*. Uma terceira fase, *Inform*, é necessária, como será visto adiante. A cada novo evento detectado, um novo assinalamento de testes é gerado, e, juntamente com o assinalamento de testes, o diagnóstico é feito.

Cada nodo tem um identificador único e um vetor *Syndromes* armazenado localmente. A entrada i do vetor contém uma lista dos nodos testados pelo nodo i e os resultados dos testes, associados a um *timestamp* local para informar a idade da informação. Nodos recebem vetores *Syndromes* provenientes de outros nodos através de pacotes ou mensagens de diagnóstico.

Pacotes são utilizados para distribuir informação entre os nodos e para indicar quando os nodos devem adaptar seu assinalamento de testes. Há três tipos de pacotes: pacotes *Search*, pacotes *Destroy* e pacotes *Inform*, correspondentes a cada uma das fases do algoritmo. Os pacotes contêm a seguinte informação: identificador do nodo originador do pacote, ou raiz, uma cópia do vetor *Syndromes* daquele nodo e uma lista de nodos que já receberam e encaminharam aquele pacote. Os pacotes fazem uma travessia distribuída de busca em profundidade pelo grafo, passando por todos os nodos sem-falha e retornando ao nodo raiz.

O algoritmo constrói um assinalamento adaptativo de testes nas fases *Search* e *Destroy*. Durante a primeira fase, que opera em paralelo em todos os nodos sem-falha do sistema, testes são adicionados localmente em cada nodo para estabelecer um assinalamento de testes fortemente conectado. Na fase seguinte, que opera de maneira sequencial, testes redundantes são removidos, resultando num assinalamento mínimo que forma um grafo fortemente conexo.

A Fase *Search* Durante a operação do algoritmo são feitos testes periódicos. Ao ser detectado um novo evento, por exemplo, de falha em um nodo, o nodo executando o teste inicia a fase *Search*. Nesta fase, todos os nodos vizinhos são examinados e um teste de um vizinho é adicionado localmente se não existir nenhum caminho para aquele nodo no assinalamento atual de testes, possivelmente devido à falha do outro nodo, recém detectada.

O algoritmo do caminho mínimo de Dijkstra [Cormen, Leiserson, Rivest *et. al.*, 2001] é, então, executado no vetor *Syndromes* corrente para detectar possíveis caminhos. Um pacote *Search*, contendo o vetor *Syndromes* atualizado é gerado e enviado a todos os nodos sem-falha.

Os demais nodos executam a fase *Search* ao receber o pacote. Se um nodo adiciona novos testes, um novo pacote *Search* é iniciado e reflete o vetor *Syndromes* novamente atualizado.

Múltiplos pacotes *Search* são progressivamente reduzidos a um único pacote final durante colisões de pacotes nos nodos da rede. Por exemplo, se um pacote B chega a um nodo que está correntemente na árvore de travessia de um pacote A, que tenha chegado antes, uma de três situações pode ocorrer: se o pacote B possui informação mais recente que o pacote A, ou B é *dominante* sobre A, ele continua a travessia; se A é *dominante* sobre B, a travessia do pacote B é descontinuada; se os pacotes são dominantes um com relação ao outro, ou *bidominantes*, então um novo pacote é iniciado naquele nodo.

O diagnóstico correto é obtido ao final da fase *Search*, após sua execução em todos os nodos sem-falha. O grafo de testes é fortemente conexo desde que a rede de nodos sem-falha esteja conectada. Se a rede estiver desconectada, o assinalamento corresponde a um grafo fortemente conexo em cada componente conectado da rede.

As Fases *Destroy* e *Inform* A fase *Destroy* é iniciada ao final da fase *Search* e remove possíveis assinalamentos redundantes introduzidos. Um assinalamento de testes é removido se outros caminhos existem para aquele nodo no assinalamento de testes corrente. O pacote utilizado contém o vetor *Syndromes* atualizado e distribui o assinalamento

mais recente a todos os nodos sem-falha.

O único pacote *Search* que completou aquela fase inicia a fase *Destroy* e esta opera sequencialmente para assegurar que o grafo de testes seja mantido fortemente conexo. Após execução em todos os nodos sem-falha, o assinalamento de testes é mínimo e corresponde a um grafo fortemente conexo.

Ao final da fase *Destroy*, embora todos os nodos já tenham diagnóstico correto, eles podem possuir *arrays Syndromes* diferentes armazenados localmente durante esta última fase. Um pacote *Inform* único, enviado pelo nodo que concluiu a fase *Destroy*, é utilizado para informar sequencialmente a todos os nodos sem-falha o assinalamento correto de testes.

2.2.2.2 Um Exemplo de Execução

O seguinte exemplo de execução do algoritmo, reproduzido aqui, é descrito no trabalho de [Stahl, Buskens e Bianchini, 1992a]. O sistema da Figura 2.4 possui um assinalamento de testes fortemente conectado. Supõe-se que o nodo 5 falha e que ambos nodo 3 e nodo 4 detectam a falha e iniciam o procedimento *Search*. Isto é mostrado na Figura 2.5.

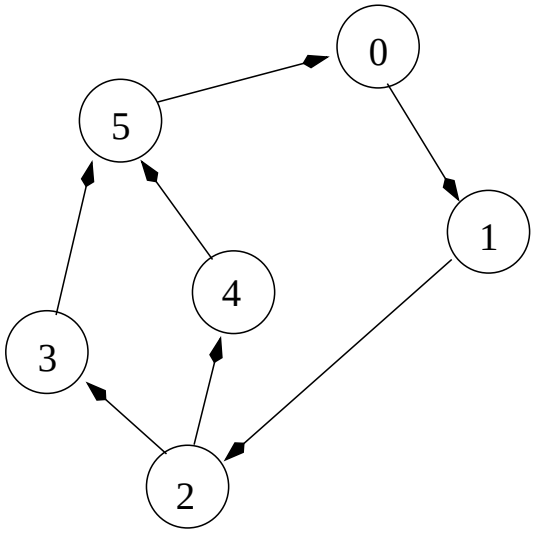

Figura 2.4: Grafo do sistema com configuração inicial de testes.

Como o grafo de testes não é mais fortemente conexo devido à falha do nodo 5, o nodo 3 adiciona um teste para o nodo 2 e o nodo 4 adiciona um teste para o nodo 0.

Ambos os nodos criam e enviam pacotes *Search* para nodos vizinhos. Assume-se que o pacote vindo do nodo 3 chega ao nodo 2 e este nodo também inicia o procedimento *Search*, não adicionando mais testes e enviando o pacote recebido. Em seguida, o pacote vindo do nodo 4 também chega ao nodo 2. Este nodo possui informação do ponto de vista do nodo 3 e recebe informação do ponto de vista do nodo 4. O pacote recebido, portanto, é *bidominante* relativamente à informação armazenada no nodo 2. Este nodo, então, não adiciona mais testes e inicia um novo pacote *Search* com a informação atualizada.

Os dois pacotes *Search* circulando colidem, então, em algum nodo e o pacote *dominante* sobrevive, completando esta fase e restaurando o diagnóstico. A Figura 2.5 mostra o assinalamento de testes ao final da fase *Search*

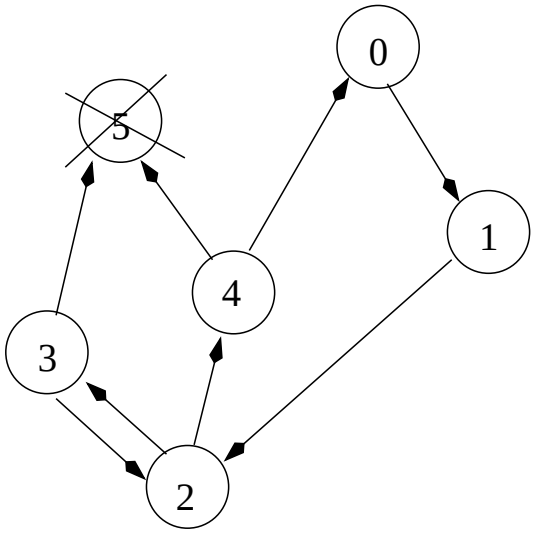

Figura 2.5: Fase *Search*.

Posteriormente, a fase *Destroy* inicia no nodo onde a fase *Search* completou, supostamente no nodo 1. Este nodo não remove testes e envia um pacote *Destroy* para o nodo 2, que também não remove testes e envia o pacote para o nodo 3 e o nodo 4, sequencialmente. Assume-se que o nodo 4 é o primeiro a receber e remove o teste redundante do nodo 5. Nenhum teste a mais é removido e o pacote visita os demais nodos, retornando ao nodo 1. Este nodo cria e envia um pacote *Inform* com o último assinalamento de testes, o qual é mostrado na Figura 2.6.

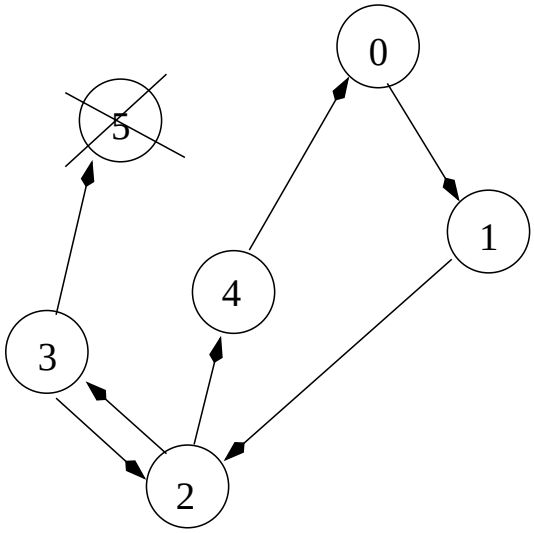

Figura 2.6: Fase *Destroy*.

Quando o nodo 5 é reparado, o nodo 3 e o nodo 5 geram pacotes *Search*. O nodo 5 adiciona um teste para o nodo 0 e os demais nodos não adicionam mais testes. Supondo que a fase *Destroy* inicie no nodo 3, este remove o assinalamento de teste para o nodo 2 e encaminha o pacote *Destroy*. O assinalamento de testes final é mostrado na Figura 2.7 e difere do assinalamento inicial.

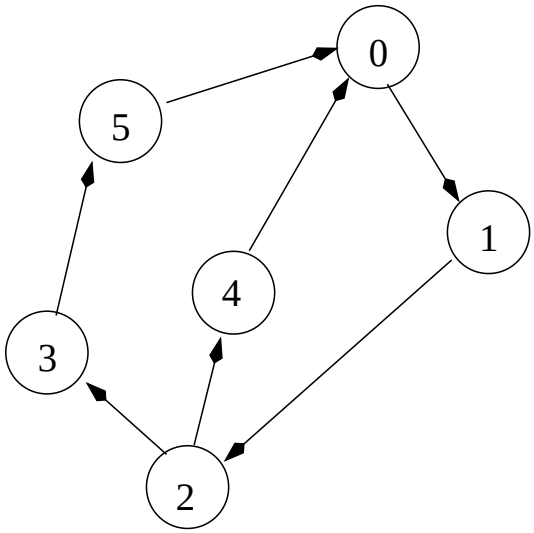

Figura 2.7: Nova fase *Search*.

A ocorrência de múltiplos eventos durante as fases de *Search*, *Destroy* ou *Inform* é tratada da seguinte maneira: a ocorrência e detecção de um novo evento gera um novo pacote *Search*, que se torna dominante e causa o progressivo abandono do pacote *Search*

anterior ou dos pacotes *Destroy* ou *Inform*, conforme a fase em que a nova disseminação se iniciou.

2.2.2.3 Desempenho e Correção do Algoritmo

Os autores provam em [Stahl, Buskens e Bianchini, 1992b] que um único pacote Search é gerado após múltiplas colisões e, portanto, um único pacote *Destroy* é iniciado, assegurando a correção do algoritmo.

Após a execução do algoritmo, o grafo de testes é minimo e fortemente conexo, e o número de testes é limitado em fases sem ocorrência de eventos, sendo maior ou igual a n e menor ou igual a $2(n-1)$, onde n é o número de nodos no sistema.

Em períodos de transição, após a ocorrência da falha ou da recuperação de um nodo, o tráfego de pacotes aumenta e a análise do desempenho do algoritmo leva em conta os melhores e piores casos da falha e da recuperação de um nodo, em termos de número de pacotes gerados nas três fases do algoritmo, número de testes gerados e latência de diagnóstico em número de rodadas. A análise detalhada pode ser consultada em [Stahl, Buskens e Bianchini, 1992a].

2.2.3 O Algoritmo RDZ

O algoritmo descrito nesta seção tem seu nome das iniciais de seus autores. Neste algoritmo, proposto por [Rangarajan, Dahbura e Ziegler, 1995], a rede tem topologia arbitrária e um nodo sem-falha sabe quais são seus vizinhos. Os nodos falham segundo o modelo *crash* e um nodo sem-falha é capaz de testar um nodo vizinho e de responder a um teste feito por um de seus vizinhos. Um nodo sem-falha pode se tornar falho e um nodo falho pode se recuperar a qualquer tempo.

O algoritmo não distingue entre uma falha em um nodo e uma falha em um canal de comunicação. Assim, o nodo que manda uma mensagem para um nodo vizinho recebe uma resposta de teste ou um *acknowledgement* dentro de um certo período somente se o outro nodo está sem-falha. Os enlaces de comunicação têm atraso limitado.

Nodos monitoram uns aos outros periodicamente e um nodo sem-falha é testado

por exatamente um outro nodo. Cada nodo sem-falha é responsável por assegurar que exatamente um de seus vizinhos sem-falha (se existe um nestas condições) o está testando. Desta forma, um nodo sem-falha pode requisitar que um nodo vizinho se torne seu testador.

Nodos falhos não são monitorados e um nodo que recupera não depende de que outros nodos detectem esta ocorrência. Um nodo recém recuperado não tem conhecimento sobre o estado dos demais nodos da rede. Quando um nodo detecta que um nodo que ele está monitorando tornou-se falho ou que um nodo falho recuperou, ele propaga esta informação para todos os seus vizinhos, que a propagam por sua vez a seus vizinhos na rede e assim por diante.

2.2.3.1 A Transação de Validação

O algoritmo executa em dois passos: detecção de falhas em nodos vizinhos e disseminação de informação sobre as mesmas. Um *evento* é definido como a transição de um nodo tanto do estado sem-falha para o estado falho como do estado falho para o estado sem-falha.

Nodos testam outros nodos periodicamente e o nodo que detecta uma falha propaga esta informação para os demais. A informação é propagada por meio de uma *transação de validação*: quando um nodo propaga informação para outro, o primeiro determina antes se o segundo está falho, e o faz enviando uma mensagem e esperando uma confirmação. Assim, existem dois mecanismos de detecção de falhas: um é através do teste propriamente dito e o outro é através de um *timeout* de uma mensagem enviada como parte de uma transação de validação. Se um nodo não responde corretamente a uma mensagem de validação, o nodo que a enviou dissemina informação sobre este evento de falha; assim, transações de validação são também utilizadas como testes.

Os dados que são enviados na mensagem da transação de validação consistem, entre outros, de um vetor de eventos, $event[n]$, onde n é o número de nodos da rede, com cada entrada contendo um contador de eventos detectados para aquele nodo. O contador de eventos é inicializado com 0 e incrementado a cada evento detectado, de forma que um contador par significa nodo sem-falha, enquanto que um contador ímpar significa nodo

falho. Este vetor é cópia de um vetor similar, mantido localmente em cada nodo.

Quando um nodo sem-falha recebe uma transação de validação, ele checa se a informação contida na mensagem é a mesma, ou considerada *antiga* ou considerada *nova*, comparada com suas próprias informações. Se a informação é a mesma, então o nodo que recebeu a mensagem tem a mesma informação que o nodo que a enviou a respeito do estado de todos os nodos do sistema. Se a informação é *antiga*, então o nodo que recebeu a mensagem tem informação mais recente a respeito de pelo menos um dos nodos e a mesma informação a respeito dos demais nodos. Ainda, se a informação recebida é *nova*, então o nodo que a recebeu tem informação mais antiga do que aquele que a enviou a respeito de pelo menos um dos nodos da rede e tem a mesma ou nova informação sobre o restante dos nodos.

Se a informação recebida é a mesma, então esta informação não é propagada adiante pelo nodo. Se a informação recebida é antiga, então a informação da mensagem é atualizada e enviada como resposta ao nodo (e somente ao nodo) que a enviou. Se a informação recebida é nova, a informação local e a informação contida na mensagem são ambas atualizadas conforme o caso e uma transação de validação é enviada a todos os vizinhos do nodo que a recebeu.

É utilizado um vetor de bits de tamanho igual ao número de nodos no sistema, no qual uma entrada é assinalada com 1 se a mensagem já visitou aquele nodo. Este vetor é utilizado para diminuir o número de mensagens redundantes e é inicializado com 0 quando um nodo forma uma mensagem para enviá-la.

2.2.3.2 Nodos Órfãos

Um nodo do sistema é dito *órfão* se não há nenhum outro nodo que seja seu testador. Isto pode ocorrer tanto para nodos no estado falho como para nodos no estado sem-falha. Se um nodo se torna falho, após a detecção deste evento o nodo que o testava deixa de testá-lo e ele se torna um *órfão falho*. Quando um nodo falho é reparado, ele requisita a cada um seus vizinhos, um por um, que o testem, até obter uma resposta positiva, ocasião em que ele deixa de estar órfão. O nodo que aceitou testar o nodo recém recuperado detecta

este novo evento e envia uma transação de validação a todos os seus vizinhos.

Se o testador de um nodo se torna falho, o nodo se torna um *órfão sem-falha* e pode não detectar imediatamente que não está sendo testado. Quando um de seus vizinhos sem-falha lhe envia uma mensagem contendo possivelmente a informação sobre a falha do nodo testador, o nodo percebe que está órfão e então solicita a um de seus vizinhos que o teste.

Um nodo órfão sem-falha pode se tornar falho antes que a informação sobre a falha de seu testador chegue até ele. Se isso ocorrer, a falha do nodo órfão não é detectada de imediato. Quando outro nodo envia a este uma mensagem sobre o novo estado da rede, por exemplo, a de que seu testador tornou-se falho, detecta a falha do nodo órfão e inicia uma disseminação sobre ela.

2.2.3.3 A Falha *Jellyfish*

Se ocorre a falha simultânea de vários nodos, vizinhos a um componente conexo de nodos sem-falha, então estas falhas serão detectadas se ao menos um dos nodos falhos for testado por um dos nodos sem-falha. A falha de um dos nodos é detectada pelo nodo testador e a falha dos demais através dos *timeouts* das transações de validação.

Entretanto, se os nodos formarem uma configuração chamada pelos autores de *jellyfish* (em português, medusa), onde há um conjunto de nodos que testam todos uns aos outros, formando um anel, e esta configuração for vizinha a um componente conexo de nodos sem-falha, em que nenhum deles testa um dos nodos da *jellyfish*, é possível que a falha simultânea de todos os nodos da configuração *jellyfish* não seja detectada.

A Figura 2.8 mostra uma configuração deste tipo, em que o nodo B e o nodo C formam a cabeça da *jellyfish*, com "tentáculos" emergindo para os nodos A e nodo D.

Os autores argumentam quanto à pequena probabilidade de formação de uma configuração *jellyfish* com vários nodos, entretanto esta configuração pode ocorrer em qualquer sistema de topologia arbitrária e corresponde efetivamente a uma situação de falha não detectável.

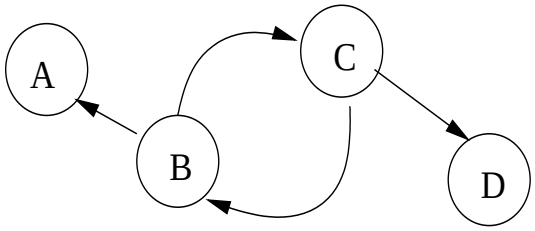

Figura 2.8: Um exemplo de *jellyfish*.

2.2.4 Os Algoritmos NBND e DNC

Esta Subseção apresenta os algoritmos *Non-Broadcast Network Diagnosis* (NBND) e *Distributed Network Connectivity* (DNC) cuja evolução levou ao trabalho apresentado nesta tese.

2.2.4.1 *Non-Broadcast Network Diagnosis*

O algoritmo NBND, ou *Non-Broadcast Network Diagnosis* foi concebido para diagnóstico de redes ponto a ponto [Duarte, Mansfield, Nanya *et. al.*, 1997, Duarte, 1998]. O algoritmo realiza o diagnóstico de *timeouts* em enlaces e calcula quais nodos da rede estão alcançáveis e quais não estão.

Para isso, todos os nodos tem conhecimento da topologia da rede e enlaces são testados periodicamente. Nodos disseminam informação de *timeouts* de enlaces para seus vizinhos sem-falha. Um novo evento somente pode ocorrer após a detecção e disseminação da informação sobre o evento anterior, isto é, não são permitidos eventos simultâneos em vários pontos da rede.

O algoritmo utiliza o número mínimo de testes por enlace, ou seja, um testador *por enlace*, e o nodo com maior identificador é o nodo testador, em cada enlace. Os testes são periódicos, em intervalos de teste.

São assumidos dois tipos de eventos: eventos de falha ou *fault events* e eventos de recuperação ou *repair events*, sempre relativos a enlaces, embora nodos também possam falhar. Os eventos são relativos a enlaces porque em redes de topologia arbitrária, como é o caso de redes ponto a ponto, é impossível distinguir um nodo falho de um nodo com todos os seus enlaces falhos. Como um nodo com todos os enlaces falhos fica inalcançável,

os nodos podem estar nos estados *sem-falha* e *inalcançável*, enquanto os enlaces podem estar nos estados *sem-falha* e *timed-out*.

Os testes de enlaces são feitos da forma dita *two-way* pelos autores. Neste tipo de teste, quando um nodo testa outro, o testador toma o tempo local do nodo testado e guarda o resultado no nodo testado. Desta forma, o nodo testado pode monitorar a atividade do nodo testador. Se o nodo testador exceder um limite relativo ao tempo máximo entre intervalos de teste, o nodo testado conclui que há um *timeout* no enlace, por falha no enlace ou no nodo testador, e passa a testá-lo. Esta configuração de testes se mantém assim até a recuperação do enlace ou do nodo de maior identificador. Então o nodo de maior identificador passa novamente a testar o enlace.

Em cada um dos nodos, um contador de estados é mantido para cada enlace do sistema. O contador é inicialmente zero e é incrementado a cada novo evento detectado ou informado para aquele enlace.

Quando da detecção de um evento em um enlace, o nodo que o detectou propaga informação sobre o mesmo para todos os seus nodos vizinhos, estes para seus vizinhos e assim sucessivamente, à semelhança do tipo de disseminação empregada no algoritmo RDZ [Rangarajan, Dahbura e Ziegler, 1995]. A mensagem de disseminação informa a identificação do enlace e o respectivo contador de eventos, além da lista com cada um dos nodos que já processou aquela mensagem, de forma a reduzir o número de mensagens redundantes.

Ao receber informação sobre um evento, cada nodo executa um algoritmo para calcular a conectividade da rede, em termos de quais nodos estão alcançáveis e quais nodos estão inalcançáveis.

O algoritmo permite eventos que particionam a rede, uma vez que um *timeout* em um enlace é percebido por ambos, nodo testado e nodo testador. Desta forma, cada um dos componentes conexos recebe informação sobre o evento. O algoritmo permite também a posterior reparação de nodos ou enlaces falhos. Eventos de recuperação de um enlace que particionava a rede em dois componentes conexos são tratados assumindo que ambos os nodos vizinhos àquele enlace trocam informação sobre eventos passados,

ocorridos posteriormente à desconexão. O algoritmo, porém, não permite a ocorrência de eventos simultâneos em vários pontos da rede.

O número mínimo de um testador por enlace por intervalo de testes é utilizado. Entretanto, isso não é assegurado para enlaces falhos com ambos os nodos sem-falha. Nestas condições, o número de testadores por enlace sobe para dois.

A latência de diagnóstico é ótima e igual ao diâmetro da rede. A disseminação utiliza mecanismos para reduzir o número de mensagens redundantes devido à estratégia paralela de disseminação e para não permitir que a mensagem percorra ciclos na rede. Os autores argumentam, ainda, que a mensagem é pequena, o que minimiza seu impacto na rede.

A distribuição de testes utilizada pelo *NBND*, em que o nodo com maior identificador em um enlace é responsável por testá-lo, pode levar a situações em que alguns nodos executam mais testes do que outros nodos da rede [Duarte, 1998]. Para resolver este problema, [Siqueira, Fabris e Duarte, 2000] propuseram uma nova estratégia de testes, baseada em *token*, de forma a dividir a responsabilidade do teste entre os dois nodos que compartilham um enlace. O teste *two-way* gera mais mensagens porque ambos os nodos de um enlace detectam um evento no mesmo. Já o teste baseado em *token* faz com que somente um dos nodos, o testador naquele momento, detecte o evento e, assim, esta estratégia de testes tem implicações também no desempenho do algoritmo em termos de número de mensagens geradas por evento detectado.

2.2.4.2 *Distributed Network Connectivity*

O algoritmo *Distributed Network Connectivity* [Duarte e Weber, 2003a] permite cálculo de conectividade de redes de topologia arbitrária. O algoritmo consiste de três fases: testes, disseminação da informação sobre novos eventos e cálculo local da conectividade. Nodos executam testes para determinar o estado de enlaces adjacentes. Se não há resposta a um dado teste sobre um enlace, o nodo testador não é capaz de determinar se o enlace testado ou o nodo conectado por aquele enlace não está operando apropriadamente. Se não há nenhuma resposta a testes executados sobre todos os enlaces de um nodo, então

o nodo é considerado *inatingível*. Nodos, portanto, podem estar nos estados *sem-falha* ou *inatingível*, e enlaces podem estar em um de três estados: *sem-falha*, *silencioso* ou *inatingível*.

A qualquer tempo qualquer nodo sem-falha executando o algoritmo pode computar localmente a conectividade da rede após remover da topologia os enlaces que estão no estado *silencioso*.

Cada enlace é testado a cada intervalo de testes. Há um testador por enlace. O algoritmo, portanto, requer o número mínimo de testes para qualquer topologia de rede. O algoritmo utiliza uma estratégia de testes baseada em *token*, proposta inicialmente por [Siqueira, Fabris e Duarte, 2000]. Ambos os nodos conectados por um enlace executam testes sobre aquele enlace em intervalos alternados. Os testes utilizados são também ditos *two-way tests*, no sentido em que, quando um nodo executa um teste sobre um enlace, não somente o nodo testador determina o estado do nodo testado, mas também o nodo testado determina o estado do testador.

Cada nodo mantém um *timestamp* que é um contador de estados para cada enlace no sistema [Rangarajan, Dahbura e Ziegler, 1995]. Cada *timestamp* é inicialmente zero, e é incrementado a cada novo evento detectado no respectivo enlace. Isto permite a um nodo identificar mensagens redundantes. Uma mensagem é considerada redundante quando não é a primeira sobre um dado conjunto de eventos. Após um novo evento ser detectado, o testador propaga a informação sobre o mesmo a seus vizinhos. Cada nodo mantém a topologia completa da rede. A estratégia de disseminação paralela empregada é baseada em uma árvore distribuída de busca em largura.

Cada mensagem de diagnóstico leva a seguinte informação: a raiz da árvore, e um conjunto de informações de eventos em enlaces que contém, para cada evento: (1) a identificação do nodo testador, (2) a identificação do nodo testado, e (3) o *timestamp* para o enlace testado. Cada nodo rodando o algoritmo mantém uma tabela de enlaces indexada pelos identificadores do enlace, contendo o *timestamp* para o enlace. Um *timestamp* par indica um enlace sem falha; um *timestamp* ímpar indica que o enlace está silencioso [Rangarajan, Dahbura e Ziegler, 1995].

Cada nodo mantém o mesmo grafo representando a topologia da rede, a qual é atualizada a cada evento nas mensagens recebidas, isto é, enlaces silenciosos são removidos do grafo. Como cada nodo também é informado sobre a raiz da árvore quando do recebimento de dada mensagem, todos os nodos montam a mesma árvore de busca em largura ao disseminar aquela mensagem.

Considerando a árvore de busca em largura, após a disseminação ter sido iniciada, as novas informações sobre eventos são consideradas *pendentes*, até que cada nodo sem-falha no sistema confirme o recebimento daquela informação. Mensagens de confirmação são propagadas dos nodos-folha para a raiz. Sempre que um nodo é uma folha, após receber a mensagem de seu pai ele envia uma confirmação, chamada *ack*.

Após receber *acks* de todos os seus filhos, um nodo envia um *ack* a seu pai na árvore. A disseminação é completada quando a raiz recebe confirmações de todos os seus filhos.

Disseminação de Múltiplos Eventos Concorrentes Quando um novo evento ocorre e a disseminação do evento precedente ainda não foi completada, nodos devem realizar a disseminação de múltiplos eventos.

Seja uma "mensagem pendente" a mensagem de uma disseminação pendente. Seja "mensagem recebida" a mensagem recebida por um nodo. Ambas as mensagens podem conter novas informações uma com relação à outra. Uma mensagem contém informação nova se ela contém informação sobre um evento em um enlace que não está presente na outra mensagem, ou se ela possui informação sobre um enlace que está na outra mensagem, porém com *timestamp* maior.

Quando a mensagem recebida tém informação nova com respeito à mensagem pendente, o nodo agrega a nova informação recebida à informação na disseminação pendente. Se a mensagem pendente tem nova informação com respeito à mensagem recebida, o nodo inicia uma nova árvore de disseminação. Se esse não é o caso, então o nodo simplesmente se insere na árvore de disseminação distribuída de acordo com a mensagem recebida. A disseminação precedente é abandonada quando a nova disseminação tem informação com-

pleta sobre todos os eventos. Se a mensagem recebida não possui informação nova com respeito à mensagem pendente, então a mensagem recebida é simplesmente descartada.

Quando a mensagem recebida e a mensagem pendente são exatamente as mesmas, elas vêm necessariamente de árvores diferentes, iniciadas por nodos diferentes. Nesse caso, o nodo tem que tomar parte em ambas as árvores de disseminação. Esta situação ocorre quando dois ou mais nodos são informados sobre o mesmo conjunto de eventos antes que a disseminação iniciada por um deles alcance os demais. Neste caso, nenhuma das árvores de disseminação pode ser abandonada, por não haver critérios para selecionar uma das mensagens e descartar as demais.

2.2.5 O Algoritmo *ForwardHeartbeat*

O algoritmo *ForwardHeartbeat* foi apresentado no contexto de *bounded correctness* pelos autores [Subbiah e Blough, 2004]. O algoritmo segue o mesmo princípio do algoritmo *HeartbeatComplete*, baseado em detecção de atrasos de *heartbeats* para obter diagnóstico das unidades do sistema. Para redes ponto-a-ponto, para as quais este algoritmo foi concebido, há a restrição de que o número de nodos falhos em qualquer instante de tempo não exceda o limite que desconecte a rede. Particionamentos da rede, portanto, não podem ocorrer.

O algoritmo também consiste no envio periódico de *heartbeats* pelos nodos sem-falha em todos os seus enlaces. Estes *heartbeats*, aos serem recebidos pelos nodos vizinhos, são encaminhados aos demais nodos e assim sucessivamente.

Os nodos mantêm um *buffer* local onde são armazenados os *heartbeats* recebidos. Quando um nodo detecta um *timeout* do *heartbeat* de outro nodo, remove o *heartbeat* deste último de seu *buffer* local e o diagnostica como falho. Quando um nodo recupera, o nodo vizinho que detectou a recuperação lhe envia os *heartbeats* mantidos em seu *buffer* local.

Uma mensagem de *heartbeat* contém os seguintes campos: *node_id*, o identificador do nodo que iniciou o *heartbeat*; *seq_no*, o número de sequência do *heartbeat*, incrementado pelo nodo que envia um *heartbeat* a cada nova mensagem gerada; e *delay*, o tempo mínimo que um *heartbeat* esteve na rede antes de ser recebido por determinado nodo.

Os dois primeiros campos identificam o *heartbeat*, e o *delay* é calculado como segue: um nodo que inicia um *heartbeat* o armazena localmente com o campo de *delay* igual a zero. Antes de enviar a mensagem, este campo recebe o tempo que o *heartbeat* permaneceu armazenado localmente mais o tempo mínimo que este *heartbeat* vai experimentar na rede até chegar ao nodo vizinho. Portanto, um nodo tem informação sobre o tempo mínimo que os *heartbeats* que ele mantém armazenados existiram na rede. Após o recebimento do *heartbeat* de determinado nodo, o período de tempo esperado para detecção de um *timeout* do seu próximo *heartbeat* é calculado levando em conta o valor armazenado no campo de *delay* relativo àquele nodo.

Localmente, cada nodo mantém um *array* de tamanho n, onde n é o número de nodos na rede. Para cada entrada do *array*, é mantido o estado que o nodo conhece para aquele nodo, como *failed*, *working* ou *unknown*. Também é mantido o campo de *seq_no* do último *heartbeat* recebido daquele nodo.

Os autores calculam os parâmetros do modelo de *bounded correctness* para que o algoritmo obtenha latência de diagnóstico e tempo de retenção de estado mínimos. Os tempos mínimo e máximo que um *heartbeat* leva para alcançar todos os nodos sem-falha também são provados. Com isso, o tempo de *timeout* para diagnosticar um nodo como falho também é obtido.

2.3 OUTRAS ABORDAGENS RELACIONADAS

Nesta Seção serão brevemente citadas outras abordagens relacionadas ao tópico tratado no presente trabalho. Serão apresentados trabalhos relacionados à área de serviços de grupo particionáveis, bem como outros modelos ou algoritmos de diagnóstico em nível de sistema. Serão mostrados também resultados de diagnóstico em nível de sistema no contexto de redes dinâmicas, neste caso redes móveis sem fio *ad hoc* e redes de sensores.

2.3.1 Serviços de Grupo Particionáveis

Um problema relacionado ao diagnóstico distribuído é o problema da gestão da composição de grupos. Um grupo dinâmico é um grupo de processos cuja composição pode mudar

durante a computação, por exemplo devido a falhas de processos. O problema, tratado por [Birman, 1993, Chockler, Keidar e Vitenberg, 2001, Schiper e Toueg, 2006], consiste em estabelecer uma visão. Os serviços de grupo utilizam detectores de defeitos e comunicação confiável em grupo para atingir este fim. Detectores de falhas são uma abstração utilizada para solucionar problemas tais como consenso distribuído e difusão atômica em sistemas assíncronos e parcialmente síncronos [Chandra e Toueg, 1996, Raynal, 2005].

Em [Hiltunen, 1995], são descritas as similaridades e diferenças entre diagnóstico distribuído e serviço de grupo. O trabalho apresenta ainda procedimentos para converter protocolos de serviços de grupo em algoritmos de diagnóstico e vice-versa. Entre outras diferenças, em diagnóstico distribuído em nível de sistema um procedimento específico de teste é desenvolvido para cada classe de objeto monitorado. Serviços de grupo, por outro lado, utilizam *heartbeats* para determinar quais processos estão falhos e quais estão sem-falha. Serviços de grupo apoiam processos de uma aplicação específica, como replicação ativa. Diagnóstico de falhas, por outro lado, é um fim em si mesmo, isto é, requer somente a descoberta de elementos falhos e sem-falha de um sistema.

Um serviço de grupo pode ser tanto com *componente primária* como *particionável*. As sucessivas composições do grupo são chamadas *visões* do grupo. Em um serviço de gestão com componente primária, uma das partições é reconhecida como a primária, e os processos podem entregar mensagens somente se pertencerem à partição primária. Neste caso, as visões instaladas por todos os processos do sistema estão totalmente ordenadas. Em um serviço particionável, todos os processos entregam mensagens, independente da partição a que pertencem. As visões instaladas, neste caso, são parcialmente ordenadas, isto é, múlitplas visões disjuntas podem existir concorrentemente [Chockler, Keidar e Vitenberg, 2001, Défago, Schiper e Urbán, 2004].

O problema de grupos partitionáveis foi tratado por [Rodrigues e Guo, 2000], [Babaoglu, Davoli e Montresor, 2001] e [Fekete, Lynch e Shvartsman, 2001], entre outros [Dolev, Malki e Strong, 1996]. Em [Rodrigues e Guo, 2000], os mecanismos de reconciliação necessários quando partições reconectam no contexto do cha-

mado *Light-Weight Group Service* (LWG) são considerados. A *Light-Weight Group Service* mapeia múltiplos grupos de usuários em um pequeno número de instâncias que implementam sincronia virtual e são chamados de *Heavy-Weight Groups* (HWG). Considera-se que sistemas assíncronos exibem particionamento devido a falhas por *crash* em roteadores e enlaces, ou ainda devido a carga excessiva em porções da rede.

Em [Babaoglu, Davoli e Montresor, 2001], é apresentada uma metodologia para desenvolvimento de aplicações tolerantes a particionamento baseada em serviço de grupo. O sistema é modelado como um conjunto de processos que se comunicam por troca de mensagens através de uma rede. Embora a recuperação de processos não seja considerada, falhas de comunicação podem ocorrer e são temporárias devido a reparos subsequentes. Desta forma, o particionamento desabilita a comunicação entre conjuntos de processos até que eles novamente reconectem.

Em [Fekete, Lynch e Shvartsman, 2001] é apresentada uma especificação formal para um serviço de grupo particionável orientado a visões e a utiliza para construir uma aplicação de *ordered-broadcast*. A aplicação reconcilia informação de diferentes visões, uma vez que, após o particionamento os processos podem continuar operando em cada visão.

Em [Julien e Roman, 2004], os autores provém a especificação de um serviço de grupos particionável para redes *ad hoc* no qual as mensagens enviadas pelos membros do grupo são garantidamente entregues em grupos logicamente conectados, onde o grafo de conexão é baseado numa noção de *distância segura*. Esta propriedade baseia-se no atraso finito na entrega de mensagens dentro de uma partição e na velocidade máxima limitada para o movimento dos nodos.

Em [Aguilera, Chen e Toueg, 1999] os autores tratam do consenso em redes assíncronas particionáveis, utilizando o detector de defeitos do tipo *heartbeat* introduzido em [Aguilera, Chen e Toueg, 1997]. Os resultados deste trabalho são estendidos para redes particionáveis onde os enlaces são unidirecionais e a rede não é necessariamente completamente conectada. Uma partição é modelada por um grafo fortemente conectado e definida como um conjunto máximo de processos que estão *mutuamente* alcançáveis um

pelo outro. Entretanto, nenhuma das partições fica isolada das demais, podendo uma partição receber mensagens de outra, ou enviar mensagens para outra partição.

Em [Temal e Conan, 2004], as funcionalidades de gerenciamento de desconexão e tolerância a falhas, para ambientes onde terminais móveis podem desconectar de sua estação base, são implementadas por detectores de falhas. Além dos detectores de falhas tradicionais, dois tipos de detectores de falhas são propostos: o detector de conectividade e o detector de desconexão.

2.3.2 Outros Modelos de Diagnóstico

Em [Caruso, Chessa e Maestrini, 2007] é apresentada uma avaliação analítica do grau de *completude* do diagnóstico em sistemas regulares. De acordo com o modelo PMC [Preparata, Metze e Chien, 1968], o diagnóstico é baseado em um conjunto de testes entre unidades adjacentes cujo resultado é chamado de *síndrome*, a qual é *decodificada* para determinar o estado das unidades do sistema. Um algoritmo de diagnóstico decodifica a síndrome e divide as unidades do sistema em um conjunto de unidades declaradas como falhas, um conjunto de unidades declaradas como sem-falha e um conjunto de unidades suspeitas, cujo estado permanece não identificado. O diagnóstico é dito *correto* se o conjunto de unidades detectadas como falhas é um sub-conjunto das unidades efetivamente naquele estado, e é dito *completo* se o conjunto de unidades suspeitas é vazio. Uma medida de *completude* de diagnóstico é dada pelo número de unidades corretamente diagnosticadas, o qual é dependente da síndrome. O pior caso de completude de diagnóstico é o número mínimo de unidades corretamente diagnosticadas em todo o conjunto de todas as síndromes que podem ser geradas a partir de um número limitado de unidades no estado falho. Dos mesmos autores, um exemplo de algoritmo para decodificar uma síndrome segundo o modelo acima é o algoritmo NDA (*New Diagnosis Algorithm*) [Caruso, Albini e Maestrini, 2003]. Motivado para o diagnóstico no processo de fabricação de *chips* VLSI, o algoritmo NDA é concebido para uso em estruturas regulares como grades *toroidais* e hipercubos.

Em [Khanna, Cheng, Varadharajan *et. alli.*, 2007], os autores apresentam

uma aplicação que realiza diagnóstico de faltas em protocolos de redes de larga escala. A aplicação considera as entidades a serem diagnosticadas como caixas pretas e observa a troca de mensagens entre elas. Um grafo causal é construído para seguir a propagação do erro e identificar sua fonte.

Em outra abordagem de diagnóstico, os resultados de testes realizados pelas unidades do sistema são avaliados por *comparação*. O resultado de todas as comparações realizadas é chamado de síndrome, e são utilizados algoritmos de diagnóstico para decodificá-las. Os primeiros modelos foram apresentados em [Malek, 1980, Chwa e Hakimi, 1981, Maeng e Malek, 1981], baseados num observador central. Modelos distribuídos foram apresentados por [Sengupta e Dahbura, 1992] e [Blough e Brown, 1999]. Este último modelo baseia-se em disseminação confiável. Nos trabalhos de [Albini e Duarte, 2001, Ziwich, Duarte e Albini, 2005], os autores apresentam uma abordagem distribuída que não se utiliza de uma primitiva de difusão confiável. Em [Yang e Tang, 2007], os autores propõem um algoritmo de diagnóstico baseado no modelo clássico proposto por [Maeng e Malek, 1981], porém com desempenho melhor que os algoritmos distribuídos prévios [Sengupta e Dahbura, 1992].

No modelo PMC, um sistema t-diagnosticável pode diagnosticar até t unidades falhas. Na abordagem de diagnóstico *probabilístico*, a diagnosticabilidade do sistema leva em conta a probabilidade de falha de uma unidade, de forma que, nesta abordagem, até mais unidades podem ser diagnosticadas do que no modelo t-diagnosticável tradicional. Trabalhos como [Blough, Sullivan e Masson, 1988], [Blount, 1977], [Fussel e Rangarajan, 1989], [Maheshwari e Hakimi, 1976] e [Pelc, 1991] podem ser citados como modelos da área de diagnóstico probabilístico. O modelo de diagnóstico *probabilístico por comparação* foi introduzido por [Dahbura, Sabnani e King, 1987].

Redes dinâmicas são aquelas cuja topologia pode mudar com o passar do tempo, não somente devido à falha ou recuperação de nodos ou enlaces, como também devido à inserção e desaparecimento destes elementos da rede. Pode-se citar como exemplo de redes com estas características as redes *peer-to-peer* descentralizadas não estruturadas [Parameswaran, Susarla e Whinston, 2001] e as redes móveis sem-fio *ad hoc*

[Chlamtac, Conti e Liu, 2003]. Algoritmos de diagnóstico distribuído para redes deste tipo também estão relacionados com DNR.

Em [Santi e Blough, 2002], é considerado um sistema de nodos distribuídos em uma rede móvel sem-fio *ad hoc*, sendo cada um capaz de comunicar-se dentro de um determinado raio, e computam o tamanho máximo do raio de transmissão de forma a assegurar que a rede resultante seja conectada. Eles também consideram a versão móvel do problema, na qual é computado o intervalo máximo de tempo que um nodo pode utilizar para se mover e manter-se conectado à rede.

Em [Elhadef, Boukerche e Elkadiki, 2007] é proposto um algoritmo adaptativo de diagnóstico distribuído baseado em comparações para redes móveis *ad hoc*, o qual baseia-se no trabalho publicado por [Chessa e Santi, 2001]. Os resultados dos testes realizados localmente pelos sensores é disseminado pela rede por meio de um árvore geradora mínima adaptável, ao contrário da estratégia empregada em [Chessa e Santi, 2001], em que as informações de diagnóstico são disseminadas por inundação.

Um protoloco distribuído para diagnóstico de falhas em redes de sensores, WSN-Diag, é proposto em [Chessa e Santi, 2002]. O protocolo provê diagnóstico correto se o número de nodos falhos não superar o limite dado pela conectividade de vértices da rede. O protocolo de diagnóstico é explicitamente iniciado por um observador externo e o nodo que o inicia dissemina uma mensagem de diagnóstico por *broadcast*. As mensagens são propagadas pela rede, construindo uma árvore de nodos sem-falha. Nodos são diagnosticados como falhos se eles não enviam mensagens dentro de um determinado limite de tempo. Quando um nodo recebe mensagens de diagnóstico de seus filhos na árvore, eles combinam esta informação com seu próprio diagnóstico local e enviam a mensagem resultante para seu pai. O diagnóstico completo do sistema é posteriormente disseminado. Os autores provam que o protocolo de diagnóstico troca o número mínimo de *bits* necessários para esta tarefa, sendo, assim, ótimo sob o ponto de vista de consumo de energia.

Em [Ding, Chen, Xing *et. al.*, 2005], os autores apresentam um algoritmo tolerante a falhas para detectar nodos falhos e a borda de propagação de um evento numa rede de sensores, baseado no fato de que ambos sensores falhos e sensores próximos a um

evento apresentam leituras atípicas. Se a mudança está presente num único sensor, este é considerado falho; caso a mudança se apresente simultaneamente em vários sensores vizinhos, então um evento ocorreu.

Em [Lee e Choi, 2007], os autores apresentam um algoritmo distribuído eficiente baseado no modelo de comparações para isolar nodos falhos numa rede de sensores, utilizando a premissa de que para cada nodo sem-falha, seus vizinhos apresentam valores de leitura similares. Falhas de comunicação e leituras incorretas são mascaradas por redundância temporal.

O termo *healing*, utilizado neste trabalho para designar eventos de recuperação em nodos ou enlaces, com a possível reconexão da rede antes desconectada, aparece na literatura com uma perspectiva um pouco diferente. *Self-healing*, no campo de redes sem-fio, designa as redes *ad hoc* descentralizadas, com capacidade de se auto-organizar e reconfigurar automaticamente sem intervenção humana. Nesta área, a queda de um enlace é vista sob a perspectiva de roteamento [Johnson, 1994], com mecanismos automáticos de busca de rotas. Sob este ponto de vista, a avaliação da rede detecta falhas nos enlaces, e a descoberta de novos enlaces propicia o estabelecimento de novas rotas [Poor, Bowman e Auburn, 2003]. Em [Tipper, Dahlberg, Shin *et. al.*, 2002], no mesmo contexto, fala-se da mobilidade, com usuários se movendo em células adjacentes, desconectando e posteriormente reconectando-se à rede.

Outras abordagens para redes *self-healing* existem, não somente no campo de redes sem fio, como o uso de redundância no enlace e a detecção de falha com cálculo de novas rotas [Ayanoglu, Gitlin e Mazo, 1993]. Em [Lumetta e Médard, 2001] são citados os anéis *self-healing* das redes *SONET*. As redes *peer-to-peer* são citadas como *self-healing* no trabalho de [Kalogeraki, Gunopulos e Zeinalipour-Yazti, 2002].

Finalmente, o algoritmo proposto no presente trabalho pressupõe o conhecimento da topologia da rede pelos nodos que o executam. Em [Nassu, Nanya e Duarte, 2007] é apresentada uma estratégia para descoberta de topologia em redes dinâmicas e descentralizadas. A estratégia proposta é distribuída e se utiliza de agentes móveis, baseados em uma metáfora de colônias de formigas, as quais se comunicam através de uma estratégia

conhecida como estigmergia.

2.4 SÍNTESE DO CAPÍTULO

Este Capítulo apresentou uma descrição da área de diagnóstico em nível de sistema. Duas famílias de algoritmos de diagnóstico podem ser consideradas: aquela dos algoritmos para redes representáveis por grafos de topologia completa e aquela dos algoritmos para grafos de topologia arbitrária. Entre os algoritmos para redes de topologia arbitrária, Bagchi e Hakimi propõem uma abordagem que somente pode ser executada *off-line*. O algoritmo Adapt executa *on line* e emprega um procedimento distribuído de assinalamento de testes, durante o qual o diagnóstico é feito. O algoritmo RDZ, também executado *on line*, produz um assinalamento ótimo de testes, isto é, cada nodo tem somente um testador, e a estratégia de disseminação de informações sobre eventos é paralela, o que leva à melhor latência possível de diagnóstico. Entretanto, existe uma configuração de falha não detectável pelo algoritmo.

O algoritmo NBND detecta *timeouts* em enlaces e propõe uma estratégia de testes que permite um testador por enlace. Informações de diagnóstico são disseminadas pela rede utilizando uma estratégia paralela. Com base nesta informação, nodos podem calcular a alcançabilidade da rede. O algoritmo só permite a ocorrência de um novo evento depois que informação sobre o evento anterior já tenha sido completamente disseminada. O algoritmo DNC também calcula a alcançabilidade da rede, mas permite a ocorrência de novos eventos em qualquer fase da execução do algoritmo e de eventos simultâneos em diversos pontos da rede. O modelo proposto prevê que porções da rede fiquem inatingíveis, entretanto a recuperação da rede ainda não é tratada. O algoritmo DNC pode ser considerado parte do trabalho desta tese, tendo o algoritmo proposto neste trabalho se desenvolvido a partir dele.

O algoritmo *ForwardHeartbeat* baseia-se em *heartbeats* para obter informação de diagnóstico dos diversos nodos da rede. Embora se assuma uma rede com topologia arbitrária, particionamentos na rede não podem ocorrer.

No Capítulo seguinte, um novo algoritmo de diagnóstico em nível de sistema para

redes particionáveis de topologia arbitrária, *Distributed Network Reachability* (DNR), é apresentado.

CAPÍTULO 3

UM NOVO ALGORITMO DE DIAGNÓSTICO PARA REDES DE TOPOLOGIA ARBITRÁRIA

Neste Capítulo é descrito o algoritmo *Distributed Network Reachability* para redes particionáveis de topologia arbitrária. Um nodo executando o algoritmo calcula a alcançabilidade da rede, isto é, o nodo determina quais nodos e enlaces são atingíveis no componente conexo ao qual pertence.

O Capítulo é organizado como segue: inicialmente é apresentado o modelo de sistema. Na Seção seguinte, o algoritmo é introduzido. As três Subseções seguintes trazem a descrição das fases do algoritmo, seguidas de sua especificação formal. As provas formais das fases de teste e disseminação concluem o Capítulo. É importante destacar que no Capítulo 4 é apresentado um outro conjunto de provas formais para o mesmo algoritmo, obtidas no arcabouço *bounded correctness*.

3.1 MODELO DE SISTEMA

Considere um sistema, também chamado de rede, com topologia arbitrária, isto é, nodos são conectados por enlaces de comunicação ponto-a-ponto, em que nem todo par de nodos é necessariamente conectado por um enlace. Em tal sistema alguns nodos precisam empregar nodos intermediários para comunicarem-se uns com os outros. Exemplos destas redes incluem tanto topologias regulares tais como hipercubos e *meshes* [Culler e Singh, 1999], como grafos irregulares, tais como os que representam backbones da Internet. Cada nodo ou enlace da rede pode estar tanto falho como sem-falha. Assume-se falhas do tipo *crash*. Enlaces sem-falha oferecem serviço confiável de comunicação, isto é, enlaces não perdem mensagens e entregam-nas de forma íntegra. Nodos ou enlaces falhos podem ser mais tarde reparados, a rede pode sofrer particionamentos e subsequentemente reconectar. Assume-se um sistema síncrono.

O sistema é representado por um grafo $S = (V(S), E(S))$ onde $V(S)$ é um conjunto de N vértices ou nodos, n_0, n_1,...,n_{N-1}, e $E(S)$ é um conjunto de arestas não direcionadas ou enlaces. Pode-se alternativamente referir ao nodo n_i como *nodo* i. A aresta que conecta o nodo i e o nodo j é representada pelo par (i, j). O nodo i e o nodo j são então ditos *vizinhos* um do outro.

Periodicamente, cada nodo sem-falha executa um procedimento de teste de forma a determinar se seus vizinhos estão respondendo ou não. Um nodo testador envia uma requisição de teste para o nodo testado. Com o recebimento da resposta correspondente pode-se concluir que o nodo testado está sem-falha. Neste caso, é possível concluir que o enlace testado também está sem-falha. Entretanto, quando não há resposta do nodo testado, é impossível para o testador determinar se é o enlace de comunicação ou o nodo testado que está falho. O teste numa rede de topologia arbitrária é portanto dito *ambíguo* [Duarte, Mansfield, Nanya *et. al.*, 1997]. A Figura 3.1 ilustra esta ambiguidade: se o nodo A envia uma requisição de teste para o nodo vizinho B e um *timeout* ocorre, então é impossível para o nodo A determinar se é o nodo B que falhou ou se é o enlace de comunicação para aquele nodo que falhou.

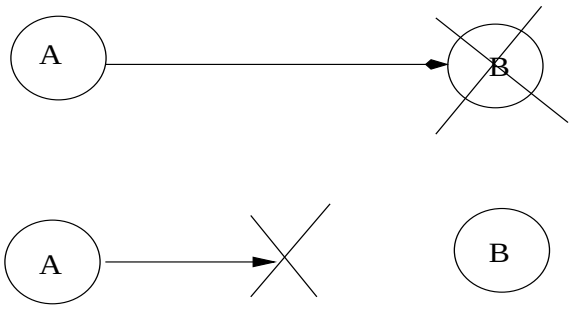

Figura 3.1: Testes são ambíguos em redes de topologia arbitrária.

Propõe-se uma estratégia para tratar este problema: quando não há resposta do nodo testado, o testador apenas conclui que o enlace testado está *não-respondendo*. O nodo correspondente pode se tornar *inatingível* se não houver outro caminho para ele.

Assim, sob o ponto de vista de um nodo testador, um *evento* é definido como uma mudança de estado de um enlace, ou de *sem-falha* para *não-respondendo* ou de *não-respondendo* para *sem-falha*. O primeiro tipo de evento é designado como um *evento de*

falha; o segundo tipo de evento é um *evento de healing* ou *de recuperação*. Embora um nodo execute testes enviando uma requisição de testes para um nodo vizinho, um evento é sempre descrito em termos do enlace de comunicação correspondente.

A partir de informações sobre estados de enlaces, os estados de nodos e de outros enlaces do sistema é calculado. Quando todos os enlaces adjacentes a um nodo estão *não-respondendo* (quer por falha do nodo, quer por falha de todos os seus enlaces), o nodo é considerado *inatingível*. Falhas de nodos ou de enlaces também podem particionar o componente. Um nodo também é considerado inatingível quando seu único enlace adjacente a um componente conexo torna-se *não-respondendo*. Enlaces de comunicação não adjacentes a nenhum nodo alcançável são também considerados *inatingíveis*. Da mesma forma, nodos não adjacentes a nenhum enlace alcançável sem falha também são considerados *inatingíveis*. Portanto, no modelo proposto, um nodo é diagnosticado como *sem-falha* ou *inatingível* e um enlace é diagnosticado em um de três estados: *sem-falha*, *não-respondendo* ou *inatingível*.

Um exemplo de rede particionável bem como dos estados de seus nodos e enlaces é ilustrado a seguir. A Figura 3.2 mostra uma rede exemplo com duas partições A e B. A rede se tornou particionada devido à falha do enlace 3–4. Tão logo os nodos 3 e 4 detectam o estado de *não-respondendo* de seu enlace adjacente, mensagens de disseminação são enviadas por ambos para cada nodo alcançável em sua partição. Suponha que, mais tarde, o enlace 1–2 também falhe e que isso seja detectado por um de seus nodos adjacentes, nodo 1 ou nodo 2. Uma mensagem de disseminação é enviada pelo nodo que detectou o evento aos nodos alcançáveis no componente A. Portanto, para os nodos no componente A, o enlace 1–2 é considerado como *não-respondendo*, enquanto para os nodos no componente B, o enlace 1–2, bem como todos os nodos e enlaces no compontente A, são considerados como *inatingíveis*.

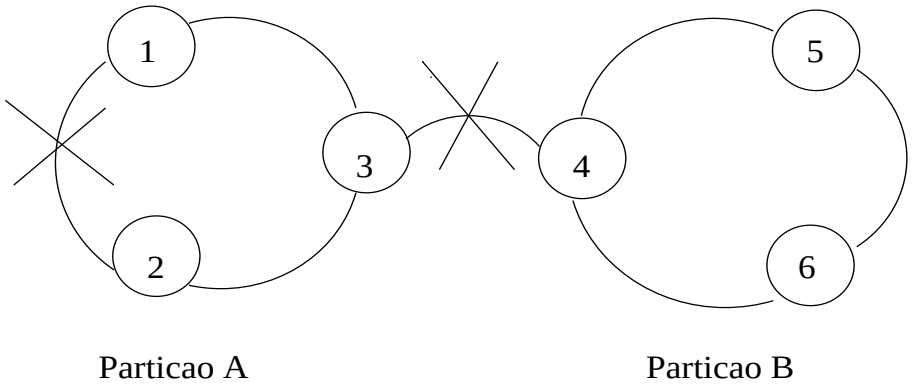

Figura 3.2: Estados de nodos e enlaces em redes particionáveis.

3.2 DESCRIÇÃO DO ALGORITMO DNR

Nesta Seção o algoritmo DNR é descrito. O algoritmo *Distributed Network Reachability* consiste de três fases: teste dos enlaces locais, disseminação de informação sobre novos eventos e cálculo local de alcançabilidade. O assinalamento de testes é dinâmico, de forma que dois nodos que compartilham um enlace de comunicação alternam-se nos papéis de *testador* e *testado* e um número ótimo de testes por enlace por intervalo de testes é assegurado: um teste por enlace por intervalo de testes, desde que ambos os nodos adjacentes estejam sem-falha.

Quando da detecção de um novo evento em um enlace local, o testador inicia a fase de disseminação, na qual uma mensagem de disseminação com informação sobre o evento é enviada a todos os nodos alcançáveis do sistema. A estratégia de disseminação utilizada é paralela e apresenta latência proporcional ao diâmetro da rede. Ambos nodos e enlaces podem falhar e recuperar durante a execução do algoritmo. A rede pode particionar e subsequentemente reconectar.

A cada vez que um nodo detecta ou recebe informação sobre um novo evento, a terceira fase é executada, na qual um algoritmo de conectividade em grafos mostra a alcançabilidade da rede do ponto de vista do nodo executando o algoritmo.

As três Subseções a seguir descrevem cada uma das fases do algoritmo, bem como apresentam sua especificação.

3.2.1 A Fase de Testes

Os testes são iniciados por um nodo, o *testador*, em outro nodo, o *testado*. Ambos o nodo testador e o nodo testado são adjacentes, isto é, eles são conectados por um enlace de comunicação, que é chamado de *enlace testado*.

Testes são executados periodicamente em intervalos de teste, que são intervalos de tempo. Dependendo da tecnologia do sistema e dos requisitos, este intervalo pode ser tão pequeno como alguns nanosegundos ou tão grande como vários minutos, ou mais. Assumindo que nodos e enlaces podem mudar de estado apenas uma vez entre dois testes consecutivos, em uma implementação os intervalos de testes devem ser estimados de forma a evitar a perda de eventos. Assume-se que os relógios são aproximadamente sincronizados, de forma que a diferença das velocidades dos relógios de quaisquer dois nodos vizinhos seja menor que o dobro uma da outra.

Os nodos mantêm uma visão local da topologia da rede representada por um grafo, no qual um *timestamp* é mantido para cada enlace. Os *timestamps* empregados são contadores de eventos similares àqueles propostos em [Rangarajan, Dahbura e Ziegler, 1995]. Os *timestamps* para todos os enlaces são inicialmente 1, isto é, todos os enlaces são inicialmente assumidos como não-respondendo. A cada vez que um novo evento é detectado, o *timestamp* para aquele enlace é incrementado. Portanto, um *timestamp* par corresponde a um enlace *sem-falha*; um *timestamp* ímpar corresponde a um enlace *não-respondendo*.

Um *assinalamento de testes* para o sistema S em um dado intervalo de testes é representado por um grafo direcionado, $T(S) = (V(S), A(S))$, onde $V(S)$ é o conjunto de vértices correspondente aos nodos da rede e $A(S)$ é o conjunto de arcos correspondendo aos testes executados. Um arco (i, j) direcionado do nodo i ao nodo j corresponde a um teste executado pelo nodo i no nodo j.

A execução de um teste do algoritmo DNR permite que ambos os nodos determinem se estão sem-falha. Como um procedimento de teste somente pode ser iniciado por um testador sem-falha e recebido através de um enlace igualmente sem-falha, quando o nodo testado recebe uma requisição de teste, ele também pode concluir que o testador está sem-falha. Portanto o nodo testado determina implicitamente o estado do testador.

Esta estratégia é chamada de *two-way test* [Duarte, Mansfield, Nanya *et. al.*, 1997]. Em um dado intervalo de testes somente um procedimento de teste deste tipo é necessário por enlace.

De forma a compartilhar as responsabilidades do teste, é proposto um assinalamento de testes baseado em *token*. Cada par de nodos conectado por um enlace compartilha um *token*; o nodo com o *token* no início de um intervalo de testes é o testador, o outro é o nodo testado. Se ambos os nodos estão sem-falha, durante o teste o *token* é transferido, e após o teste ser concluído, o nodo testado detém o *token* para o próximo intervalo de testes. Portanto, no próximo intervalo de testes os papéis são invertidos.

Se ambos os nodos e seu enlace correspondente se mantêm sem-falha, o processo é repetido, garantindo que somente um teste seja executado por enlace por intervalo de testes. Porém, a estratégia proposta também considera as situações em que o enlace, um dos nodos ou ambos os nodos se tornam falhos. Estes casos são descritos a seguir.

No caso em que um nodo se torna falho, este pode ser tanto o nodo testado ou o testador para o próximo intervalo de testes. Se o nodo testado falha, então após o testador enviar uma requisição de teste, não haverá resposta. O testador se manterá testando a cada dois intervalos de teste, como se seu vizinho estivesse sem-falha. Tão logo o nodo testado recupere, ele cria *tokens* e, após concluir a inicialização, testa todos os seus vizinhos. Os vizinhos sem-falha respondem aos testes, e obtém o *token* para o próximo intervalo de testes. Desta forma, o número de testes é reduzido quando um dos nodos está falho. Porém, tão logo o nodo falho recupere, os testes passam novamente a ocorrer uma vez por enlace por intervalo de testes.

Por outro lado, se o testador para o próximo intervalo de testes falha, o *token* desaparece. A estratégia proposta faz com que o nodo testado detecte que não recebeu uma requisição de teste e, no intervalo de testes seguinte, um *token* seja criado e o nodo assuma o papel de testador, executando um teste a cada dois intervalos de teste. Quando o nodo falho recupera, o procedimento é similar ao descrito acima. Após o período de inicialização, aquele nodo testa todos os seus vizinhos. Os vizinhos sem-falha respondem aos testes, desta forma distribuindo os *tokens* apropriadamente.

No caso de ambos os nodos falharem, nenhum teste é executado no enlace. Entretanto, uma situação peculiar ocorre quando ambos os nodos recuperam praticamente ao mesmo tempo e se tornam testadores: ambos criam *tokens* e podem executar testes *simultâneos*. Uma situação similar ocorre quando um nodo falho recupera e testa um vizinho exatamente ao mesmo tempo em que é testado por aquele vizinho. Se estes casos não são tratados apropriadamente, então ambos os nodos se tornarão testadores no mesmo intervalo de testes, e então nodos testados no intervalo seguinte, e desta forma indefinidamente. Porém, se ambos os nodos testam ao mesmo tempo, ambos recebem, um do outro, uma requisição de testes. Este fato é usado para detectar a situação e um critério baseado no identificador dos nodos é utilizado para determinar qual nodo responderá ao teste, assumindo o papel de nodo testado.

Finalmente, no caso de o enlace se tornar falho, o nodo que tem o *token* testa no intervalo de teste corrente. Um *timeout* ocorre e, se o enlace permanecer falho, no intervalo subsequente aquele nodo detecta que não foi testado e cria um *token* para o próximo intervalo de teste. O outro nodo procede de maneira similar: no intervalo de testes corrente ele detecta qeu não foi testado e cria um *token*, tornando-se o testador para o intervalo seguinte. Após a recuperação do enlace, o primeiro teste feito por um dos nodos faz com que o outro responda e se torne o testador para o intervalo seguinte. Testes simultâneos também podem ocorrer e são solucionados como descrito acima.

3.2.1.1 Especificação da Fase de Testes

Nesta subseção é apresentada a especificação da fase de testes do algoritmo. A estratégia de testes é mostrada nas Figuras 3.3, 3.4 e 3.5. A Figura 3.3 contém as declarações de variáveis e a inicialização do sistema. A estrutura de dados usada para armazenar informações sobre um evento (`Event`) é declarada. Um evento é descrito por um identificador de enlace (`LinkId`), isto é, o identificador do nodo testador, o identificador do nodo testado e o `Timestamp` correspondente. `LinkStateTable` é a estrutura que mantém a visão local da topologia, em cada nodo. `Token[j]` e `TokenTurn[j]` são as variáveis usadas para controlar a troca do *token*. `TestRequestSent[j]` e `TestRequestReceived[j]` são usadas

para detectar a simultaneidade de testes. O *timer* `TestingInterval[j]` é usado para controlar a duração do intervalo de testes para cada um dos vizinhos de um nodo, enquanto o *timer* `TestTimeout[j]` controla a duração de um teste. O *timer* `RecoveryWaitTime` será descrito a seguir.

Uma vez ocorrida uma transição do estado sem-falha para o estado falho ou vice-versa, cada nodo ou enlace deve permanecer no novo estado durante um determinado intervalo de tempo de forma que aquele evento seja detectado. Este tempo de permanência em um estado leva em conta o intervalo de testes do algoritmo e é formalizado no Capítulo 4. Entretanto, de forma a permitir que nodos se mantenham falhos por períodos muito curtos de tempo, mesmo assim garantindo que estes estados sejam adequadamente detectados, é utilizado um intervalo de tempo chamado de *recovery wait time* [Subbiah e Blough, 2004], que traduzimos por *tempo de espera para recuperação*. Durante este intervalo, o nodo que está recuperando tanto não envia requisições ou respostas a testes como não processa mensagens de disseminação.

Como pode ser visto na Figura 3.3, quando um nodo é inicializado pela primeira vez ou recupera, o *timer* `RecoveryWaitTime` é inicializado. O nodo atrasa o envio de requisições de teste até que este temporizador expire. Entretanto, como inicialmente é atribuído o valor TRUE à variável `Token[j]`, ao final do *tempo de espera para recuperação* o nodo é forçado a testar todos os seus vizinhos.

Na Figura 3.4, o módulo `TokenTest(j)` mostra a estratégia para alternar o *token*. Se um nodo tem o *token* para testar um vizinho j quando o intervalo de teste para aquele nodo expira, então o módulo `RunTest(j)` é executado. O *token* é liberado e `Token[j]` se torna FALSE. Entretanto, se o enlace está falho e a transmissão do *token* não ocorreu no intervalo de testes precedente, `Token[j]` é FALSE e `TokenTurn[j]` também é FALSE. Neste caso, ao executar `TokenTest(j)`, o valor *true* é atribuído à variável `TokenTurn[j]` para garantir que aquele nodo se torne o testador no intervalo de testes seguinte. Quando `Token[j]` é FALSE mas `TokenTurn[j]` é TRUE, um novo *token* é criado e `TokenTest(j)` é chamado recursivamente.

O módulo `RunTest(j)`, chamado pelo nodo que detém o *token*, executa o proce-

Distributed Network Reachability Algorithm Executed by Node i

```
define NodeId = node's network address;
       Timestamp = counter;
       LinkId = (NodeId, NodeId);
       Event = (LinkId, Timestamp);
       Message = list of Event;

var LinkStateTable: array[LinkId] of Timestamps;
    Token, TokenTurn: array[NodeId] of Boolean;
    TestRequestSent, TestRequestReceived: array[NodeId] of Boolean;
    TestingInterval, TestTimeout: array[NodeId] of Timers;
    RecoveryWaitTime: Timer;

Start&RunForever()
for all links do LinkStateTable[LinkId] = 1; end for
for each neighbor j
    Token[j] = TRUE;
    TokenTurn[j] = FALSE;
    TestRequestSent[j] = FALSE;
    TestRequestReceived[j] = FALSE;
    start_timer RecoveryWaitTime;
end for
when RecoveryWaitTime timer expires
  for each neighbor j
      expire_timer TestingInterval[j];
  end for
end when
while TRUE do
  for each neighbor j
      when TestingInterval[j] timer expires
           restart_timer TestingInterval[j];
           TokenTest(j)
      end when
  end for
end while
```

Figura 3.3: Algoritmo DNR: Inicialização.

Distributed Network Reachability Algorithm Executed by Node i
Testing Phase

```
|| Module: TokenTest(j)
  if Token[j]
  then Token[j] = FALSE;
       RunTest(j);
  else if TokenTurn[j]
       then TokenTurn[j] = FALSE;
            Token[j] = TRUE;
            TokenTest(j);
       else TokenTurn[j] = TRUE;
       end if
  end if

|| Module: RunTest(j)
  if (LinkStateTable[i,j] mod 2 != TestRequest(j)) /* new event detected */
  then if (LinkStateTable[i,j] mod 2 != 0) /* healing event */
       then update LinkStateTable with new info;
            LinkStateTable[i,j]++;
            Event-Disseminate(LinkStateTable); /* send timestamps > 1 */
       else LinkStateTable[i,j]++;
            Event-Disseminate(msg with new info);
       end if
       run local graph connectivity algorithm;
       for all unreachable links do LinkStateTable[LinkId] = 1; end for;
  end if

|| Module: TestRequest(j)
  send(j, test request);
  TestRequestSent[j] = TRUE;
  TestRequestReceived[j] = FALSE;
  start_timer TestTimeout[j];
  when the test reply from j arrives
       reset_timer TestTimeout[j];
       return 0; /* healed */
  end when
  when TestTimeout[j] expires
       if TestRequestReceived[j]
       then Token[j] = TRUE;
       end if
       return 1; /* failed */
  end when
```

Figura 3.4: Algoritmo DNR: Fase de testes executada pelo nodo testador.

dimento de testes e inicia a disseminação de uma mensagem contendo informações sobre o evento detectado. No caso de um evento de falha, informações sobre este único evento constarão na mensagem. No caso de um evento de *healing*, informações sobre todos os enlaces com *timestamps* maiores que 1 estão contidas na mensagem. Além disso, a terceira fase do algoritmo é executada, na qual um algoritmo de conectividade em grafos computa a alcançabilidade da rede. O valor 1 é atribuído aos *timestamps* dos enlaces tidos localmente como *inatingíveis*, de forma a não serem disseminados como eventos espúrios no caso da ocorrência de um *healing*.

`TestRequest(j)` descreve o procedimento de testes, onde um *timer* `TestTimeout[j]` é iniciado e uma requisição de testes é enviada para o nodo testado *j*. As variáveis `TestRequestSent[j]` e `TestRequestReceived` são atribuídas a cada vez que uma requisição de testes é enviada, de forma a permitir a detecção de testes simultâneos, como é mostrado a seguir.

Na Figura 3.5, o módulo `TestReply(j)` corresponde ao procedimento executado pelo nodo testado quando ele responde a uma requisição de teste. O *token* é trocado, fazendo `Token[j]` se tornar TRUE para o intervalo de testes seguinte.

Além disso, o teste *two-way* ocorre: a chegada de uma requisição de testes num enlace considerado como *não-respondendo* determina, para o nodo testado, a detecção de um evento de *healing* naquele enlace. Informações sobre a `LinkStateTable` do nodo testado são enviadas para o nodo testador como resposta ao teste. Finalmente, o *timer* `TestingInterval[j]` é reinicializado.

Para a configuração específica na qual um enlace pode ser testado simultaneamente por ambos os nodos, a variável `TestRequestSent[j]` é checada. Se um teste foi enviado para o nodo *j* e não foi ainda respondido, e.g., `TestRequestSent[j]` é TRUE, um critério é utilizado, baseado nos identificadores dos nodos. O nodo com o identificador menor conclui o teste, reinicializando ("resetando") o *timer* `TestTimeout[j]` do nodo *j* e enviando uma resposta. A detecção ocorre porque se ambos os nodos iniciaram um teste, ambos executam o procedimento `TestReply(j)` e testam o valor de `TestRequestSent[j]`.

Excepcionalmente, a requisição de testes enviada por um dos nodos chega ao vizi-

Distributed Network Reachability Algorithm Executed by Node i
Testing Phase

```
|| Module: TestReply(j)
  when a test request from j arrives
      Token[j] = TRUE;
      TokenTurn[j] = FALSE;
      if TestRequestSent[j] /* simultaneous testing */
      then TestRequestSent[j] = FALSE;
           if i > j
           then Token[j] = FALSE; /* it's i's turn to test */
                TestRequestReceived[j] = TRUE;
                return;
           else reset_timer TestTimeout[j];
           end if
      end if
      restart_timer TestingInterval[j];
      if (LinkStateTable[i,j] mod 2 != 0) /* healing event detected */
      then send(j, test reply with LinkStateTable); /* send timestamps > 1 */
      else send(j, test reply);
      end if
  end when
```

Figura 3.5: Algoritmo DNR: Fase de testes executada pelo nodo testado.

nho quando aquele nodo está no final do tempo de espera para recuperação. Chame-se de A o nodo que enviou a requisição de testes e de B o outro nodo. No caso de o vizinho B terminar o tempo de espera para recuperação e também enviar uma requisição de testes que é recebida por A antes daquele nodo detectar um *time-out* no enlace, os testes simultâneos são detectados por A, mas não por B. Se o nodo A, cuja requisição de testes foi perdida, tem o maior identificador, isto é, ele age como o testador no caso de detecção de testes simultâneos, então este nodo detecta um *time-out*, devido ao fato de sua requisição de testes ter sido perdida. O outro nodo também detecta um *time-out*, pois o nodo A não responderá à requisição de teste na tentativa de resolver a simultaneidade de testes. Assim, ambos os nodos detectam um *time out*, embora um evento de *healing* tenha ocorrido. De forma a evitar esta situação, o nodo com o maior identificador atribui TRUE à variável `TestRequestReceived[j]` quando da detecção de testes simultâneos. Se aquele nodo detecta um *time-out*, nesta ocasião ele atribui TRUE à variável `Token[j]` de forma a testar novamente após um intervalo de testes. Naquela oportunidade, o evento de *healing* é detectado. Esta é a única situação em que um mesmo nodo testa em dois intervalos de teste consecutivos. Se, por outro lado, o nodo que designamos por A tiver o

menor identificador, ao detectar a simultaneidade de testes ele responde à requisição de testes e, embora a sua requisição de testes tenha sido perdida, a simultaneidade se resolve.

3.2.2 A Fase de Disseminação

A fase de disseminação é iniciada após a detecção de um novo evento. Um evento corresponde a um enlace adjacente *sem-falha* que se torna *não-respondendo* ou vice-versa. A estratégia de disseminação empregada é paralela, iniciada pelo testador que detectou o novo evento. Este nodo monta uma mensagem de disseminação contendo informação de diagnóstico. Cada mensagem de diagnóstico contém: (1) o identificador do nodo testador, (2) o identificador do vizinho testado, e (3) o *timestamp* do enlace testado.

O nodo que inicia a disseminação envia a mensagem correspondente através de todos os seus enlaces. Um nodo que recebe uma mensagem de disseminação verifica se a mesma contém nova informação ou não. A informação é nova quando os *timestamps* de eventos contidos na mensagem são maiores que os *timestamps* correspondentes armazenados na tabela de enlaces local. Se a mensagem realmente contém nova informação, aquela informação é mantida na mensagem a ser disseminada. Informação antiga, se existente, é removida. O nodo, então, envia a mensagem através de todos os seus enlaces, exceto os enlaces por onde a mensagem chegou, e desta forma sucessivamente até que a disseminação alcança todos os nodos.

3.2.2.1 Múltiplos Eventos no Decorrer de uma Disseminação

Após uma disseminação ser iniciada e antes que ela complete, novos eventos podem ocorrer. Um evento de falha em nodo ou enlace pode confinar a disseminação a um ou mais componentes conexos. Também a ocorrência e detecção de um evento de *healing* pode ter o efeito oposto: a mensagem alcançará porções previamente inalcançáveis da rede.

Por outro lado, um nodo pode enviar uma mensagem através de um enlace *não-respondendo*, ainda considerado *sem-falha*, e detectar um novo evento de falha durante a disseminação. Como efeito colateral, este tipo de detecção reduz a latência da fase de testes.

Além disso, se duas ou mais disseminações iniciam concorrentemente em diferentes pontos do mesmo componente conexo, elas podem se encontrar em algum ponto da rede e então prosseguir e completar independentemente. O algoritmo funciona corretamente em todas estas situações.

3.2.2.2 Particionamentos da Rede e Eventos de Recuperação

Se nodos ou enlaces falham, a rede pode particionar em dois ou mais componentes conexos. Em cada componente conexo recém-formado, o estado de *não-respondendo* de um ou mais enlaces será detectado tão logo o nodo adjacente àquele enlace naquele componente conexo se torne testador. Assim que cada um dos nodos vizinhos a um enlace *não-respondendo* detecta o evento, uma mensagem de disseminação é enviada com informação sobre o mesmo a todos os nodos alcançáveis em cada partição. Quando um nodo sem-falha recebe informação sobre um enlace *não-respondendo*, ele computa a alcançabilidade da rede e atribui *timestamps* iguais a 1 a todos os enlaces inalcançáveis.

Um enlace ou nodo falhos que recuperam podem fazer com que duas ou mais partições se reconectem em um único componente conexo. Quando as partições são inicialmente reconectadas, os nodos em cada componente devem levar em conta os eventos que ocorreram nos componentes previamente inatingíveis. A passagem de um enlace do estado de *não-respondendo* para o estado *sem-falha* é chamada de *evento de healing*. A recuperação de um nodo causa eventos de *healing* em todos os seus enlaces.

Um evento de *healing* é detectado pela chegada de uma requisição de teste sobre um enlace previamente considerado como *não-respondendo*. Com o teste *two-way*, o nodo testado detecta que o enlace está sem-falha e responde com uma mensagem que contém as entradas da tabela de enlaces cujos *timestamps* são maiores do que 1, isto é, as entradas referentes aos enlaces do componente conexo ao qual pertencia previamente ao *healing*. Esta mensagem é chamada uma *mensagem de healing*. Quando a mensagem de *healing* é recebida, o testador atualiza sua tabela de enlaces local com as novas informações, incrementa o *timestamp* do enlace recuperado e inicia a disseminação de informações sobre todos enlaces com *timestamps* maiores que 1 aos seus vizinhos, incluindo o nodo testado.

Esta estratégia permite que ambos os nodos que detectaram o evento de *healing* troquem informações sobre os componentes aos quais pertenciam, de forma que *timestamps* atualizados de enlaces previamente inalcançáveis atinjam todo o novo componente conexo formado.

Como a mensagem de disseminação iniciada pelo nodo testador em resposta à detecção do evento de *healing* contém informações sobre enlaces pertencentes a ambos os componentes conexos prévios, esta mensagem de disseminação contém informações redundantes conforme a região da rede onde é disseminada. Ao receberem esta mensagem de disseminação, os vizinhos do nodo testador removem da mensagem recebida as informações redundantes em comparação com as informações armazenadas em suas tabelas de enlaces locais. Desta forma, em cada região correspondente a um dos componentes conexos prévios, só são disseminadas informações sobre o componente conexo oposto, mais o *timesptamp* atualizado do enlace que sofreu *healing*.

No caso de um enlace que se torna *não-respondendo* mas não particiona a rede, a mensagem de *healing* enviada pelo nodo testado não conterá nova informação para o nodo testador, bem como a mensagem de disseminação iniciada por este nodo, com exceção da informação sobre o evento de *healing*. Com a eliminação das informações redundantes pelos vizinhos ao nodo que deu início à disseminação, somente a informação sobre o enlace que sofreu *healing* permanece para disseminação no restante da rede.

Eventos múltiplos de *healing* ocorrem quando um nodo recupera, pois cada enlace que também não esteja falho, adjacente ao nodo que recuperou, deixa de estar no estado *não-respondendo*. Estes eventos são detectados tão logo o nodo que recuperou envia requisições de teste aos seus vizinhos. Os eventos são também detectados pelos vizinhos e o nodo que recuperou recebe mensagens de *healing* daqueles nodos. Mensagens de disseminação são criadas apropriadamente.

Se um nodo falha e recupera muitas vezes enquanto seus vizinhos permanecem no estado *sem-falha*, então estes nodos incrementam sucessivamente os *timestamps* dos enlaces para o nodo falho. Quando aquele nodo recupera, ele deve receber informação da tabela de estados de seus vizinhos antes de incrementar os *timestamps* de seus enlaces adj-

centes. Além disso, no caso de testes simultâneos, um nodo pode falhar exatamente após enviar uma requisição de teste para outro nodo que está recuperando. Se este nodo que está recuperando assume então o papel de nodo testado para aquele enlace, sua resposta ao teste chega ao testador quando aquele nodo está falho e o procedimento de *healing* não se completa. Portanto, o *timestamp* para aquele enlace não deve ser incrementado, a não ser pelo nodo testador, antes de disseminar esta informação. Um outro motivo para o *timestamp* do enlace que sofreu *healing* não ser incrementado pelo nodo testado é que este nodo, ao receber a mensagem final de *healing* enviada pelo testador, retira dela a informação redundante antes de disseminá-la nos enlaces que faziam parte de seu componente conexo. Se o *timestamp* do enlace que recuperou já tiver sido incrementado, ele é retirado da mensagem, e os demais nodos não recebem esta informação.

3.2.2.3 Especificação da Fase de Disseminação

Esta subseção traz a especificação formal da fase de disseminação do algoritmo. A fase de disseminação do DNR é mostrada na Figura 3.6. A figure 3.3 contém a declaração de variáveis globais e a inicialização do sistema. Uma mensagem (`Message`) consiste de uma lista de eventos (`Event`). Nenhum identificador de mensagem é necessário, pois a lista de eventos é suficiente para distinguir cada mensagem das demais. `LinkStateTable` é a estrutura utilizada para manter a visão local da topologia. Os `Timestamps` são inicialmente 1 e são atualizados a cada novo evento detectado em um enlace adjacente ou informado em uma mensagem de disseminação.

A fase de disseminação é iniciada quando um novo evento é detectado (módulo `RunTest(j)`). A rotina `Event-Disseminate(msg)` é chamada e recebe como parâmetro a mensagem a ser disseminada. No caso de um evento de falha, um único evento é disseminado. No caso de um evento de *healing*, informações sobre enlaces mantidas na tabela de enlaces `LinkStateTable` são enviadas para os nodos vizinhos. A `LinkStateTable` é previamente atualizada com informação enviada pelo nodo testado quando ocorre a detecção do evento de *healing*. Como é sempre o caso quando um novo evento é detectado, o algoritmo local de cálculo de alcançabilidade é executado. Aos *timestamps* dos enlaces

Distributed Network Reachability Algorithm Executed by Node i
Dissemination Phase

```
|| Module: Event-Disseminate(msg)
  for each node j neighbor of node i
      if (j != msg's senders)
      then if ((LinkStateTable[i,j] mod 2 == 0) and (send(j, msg) does not succeed)) /* fault event detected */
            then LinkStateTable[i,j]++;
                 Event-Disseminate(msg with new info);
                 run local graph connectivity algorithm;
                 for all unreachable links do LinkStateTable[LinkId] = 1; end for
            end if
      end if
  end for

|| Module: Receive(msg)
  when a dissemination message from j arrives
      if (msg has new information)
      then update LinkStateTable with new info;
            Event-Disseminate(msg with new info);
            run local graph connectivity algorithm;
            for all unreachable links do LinkStateTable[LinkId] = 1; end for
      end if
  end when
```

Figura 3.6: Algoritmo DNR: Fase de Disseminação.

no estado *inatingível* é atribuído o menor *timestamp* possível, de valor 1.

`Event-Disseminate(msg)` envia a mensagem a cada vizinho. Entretanto, durante o envio de uma mensagem um evento de falha pode ser detectado, o qual dá início a uma nova disseminação. Neste caso, após a `LinkStateTable` ser atualizada, uma nova *thread* é criada para executar `Event-Disseminate(msg)`.

`Receive(msg)` descreve o critério utilizado para dar continuidade a uma disseminação. O recebimento de nova informação é verificado. Informação antiga, se existente, é descartada da mensagem. Isto é baseado na premissa de que esta informação já foi disseminada pelo nodo que a recebeu. Após a `LinkStateTable` ser atualizada, `Event-Disseminate(msg)` é chamada. Neste caso, a mensagem é encaminhada em todos os enlaces adjacentes, exceto nos enlaces por onde ela foi recebida. O algoritmo local de alcançabilidade em grafos é executado.

3.2.3 A Fase de Cálculo de Alcançabilidade da Rede

Finalmente, a fase de cálculo de alcançabilidade da rede envolve executar, na topologia local, um algoritmo de conectividade em grafos, como o algoritmo do caminho mínimo de Dijkstra [Cormen, Leiserson, Rivest *et. al.*, 2001]. Cada nodo faz o cálculo de alcançabilidade da rede a qualquer tempo e de maneira independente dos demais.

Durante a fase de testes, se um nodo testador recebe uma resposta a uma requisição de testes, tanto o nodo testado como o enlace que os une são considerados como *sem-falha*. Se ocorre um *time-out*, o enlace é considerado como *não-respondendo*. Desconsiderando os enlaces no estado *não-respondendo* ao executar o algoritmo de conectividade, cada nodo obtém o componente conexo ao qual pertence. Se um nodo estiver falho, todos os enlaces vizinhos a ele estarão no estado *não-respondendo* e o nodo é considerado *inatingível*. Nodos e enlaces que não pertencem ao componente conexo de determinado nodo são considerados todos no estado *inatingível*.

Assim, nodos podem estar nos estados *sem-falha* ou *inatingível*. Enlaces podem estar nos estados *sem-falha*, *não-respondendo* ou *inatingível*. Em casos de particionamento, nodos situados em componentes conexos diferentes terão visões diferentes da mesma rede.

3.3 PROVAS FORMAIS DE CORREÇÃO DO ALGORITMO

Esta Seção contém as provas formais do algoritmo, tanto da fase de testes como da fase de disseminação. A correção do procedimento de *healing* também é provada.

3.3.1 Provas da Fase de Testes

Nesta Subseção são provados tanto a correção da fase de testes como o pior caso de sua latência de detecção. Demonstra-se que o algoritmo é ótimo: mesmo que dois nodos recuperem e, após o período de inicialização, eles testem um ao outro simultaneamente, o algoritmo garante que a partir do intervalo de testes seguinte os nodos se alternam testando o enlace adjacente em intervalos de testes sucessivos. Se dois nodos sem-falha estão conectados por um enlace também sem-falha, então o algoritmo garante que somente

um teste é executado por enlace por intervalo de testes.

A prova está organizada como segue. Primeiramente demonstra-se que após o período de inicialização, um nodo que recuperou testa todos os seus enlaces adjacentes uma vez a cada dois intervalos de teste. Então é demonstrado que após um evento ocorrer em um enlace adjacente a um nodo sem-falha, o nodo continua testando o enlace a cada dois intervalos de testes. Estes resultados são utilizados para provar o pior caso de latência de detecção de eventos do algoritmo.

DEFINIÇÃO 1. Um enlace *não-respondendo* adjacente a um nodo *sem-falha* faz *healing* se o nodo vizinho conclui o período de inicialização após recuperar, enquanto o enlace permanece *sem-falha*, ou se ambos os nodos permanecem *sem-falha* enquanto seu enlace adjacente *não-respondendo* recupera. O evento correspondente é chamado de um evento de *healing*.

O algoritmo de teste é baseado em uma mensagem de controle chamada de *token*, a qual é *trocada* por nodos no estado *sem-falha* através de um enlace também no estado *sem-falha* em intervalos de testes sucessivos. O *intervalo de testes* é um intervalo de tempo após o qual um determinado nodo que detém o *token* executa um teste. Nodos não estão sincronizados uns com os outros, e não compartilham um relógio global. O testador reinicializa seu intervalo de testes após enviar uma requisição de teste. O nodo testado também reinicializa seu intervalo de testes quando recebe uma requisição de teste.

Tão logo um nodo no estado *sem-falha* executando o DNR envia uma requisição de testes, ele *libera* o *token* e se torna o nodo testado para o próximo intervalo de testes. De maneira análoga, tão logo um nodo no estado *sem-falha* executando o DNR recebe uma requisição de teste, ele responde e *obtém* o *token*, se tornando o testador para o próximo intervalo de testes. O testador também libera o *token* se ele envia uma requisição de teste e o nodo testado não responde, isto é, se o enlace adjacente está *não-respondendo*. Um novo *token* é *criado* se nenhum *token* é recebido em dois intervalos de testes consecutivos.

Assume-se que as velocidades dos relógios de quaisquer dois nodos vizinhos é menor que o dobro uma da outra. Adiante será demonstrado o que ocorre se a velocidade de um dos relógios é duas vezes, ou mais, maior que a do outro.

LEMA 1. Após um nodo concluir o período de inicialização que se segue à sua recuperação e testar um enlace adjacente como *não-respondendo*, o nodo continua testando aquele enlace uma vez a cada dois intervalos de testes, desde que o enlace se mantenha *não-respondendo*.

PROVA. De acordo com a especificação do algoritmo, uma vez que um nodo conclui o período de inicialização após recuperar, ele cria um *token* para cada um de seus vizinhos e os testa. Após testar um enlace adjacente, o nodo libera o *token* correspondente e se torna o nodo a ser testado no intervalo de testes seguinte. Se o enlace para aquele nodo se mantém *não-respondendo*, o nodo não recebe uma requisição de testes e o *token* não é trocado entre eles. O nodo então atribui o valor TRUE à variável `TokenTurn` significando que um *token* deve ser criado no intervalo de testes sequinte. Portanto, no intervalo de testes seguinte um *token* é criado e o nodo testa e libera o *token* novamente. Daquele intervalo de testes em diante, o mesmo padrão de comportamento é repetido, desde que o enlace se mantenha *não-respondendo*. Portanto, o nodo mantém-se testando o enlace uma vez a cada dois intervalos de testes. □

DEFINIÇÃO 2. Dois nodos são ditos *testadores simultâneos* ou *simultaneamente testadores* quando ambos estão conectados por um enlace no estado *sem-falha* e detêm *tokens* no mesmo intervalo de testes.

DEFINIÇÃO 3. Dois testes são ditos *simultâneos* quando dois testadores simultâneos ou dois nodos simultaneamente testadores enviam requisições de teste um para o outro antes que cada um deles receba uma requisição de teste do outro.

É importante notar que testadores simultâneos não executam testes simultâneos se a requisição de teste de um deles chega ao outro antes que a requisição de teste daquele seja enviada.

O lema seguinte mostra as situações nas quais testadores simultâneos podem ocorrer.

LEMA 2. Quando um nodo termina o período de inicialização após recuperar e um enlace adjacente está *sem-falha*, se o vizinho correspondente está ou concluindo o período de inicialização após recuperar ou já está no estado *sem-falha*, estes dois nodos podem se

tornar testadores simultâneos. Testadores simultâneos também podem ocorrer se ambos os nodos estão no estado *sem-falha* enquanto seu enlace adjacente que estava no estado *não-respondendo* recupera. Testadores simultâneos também podem ocorrer se dois nodos no estado *sem-falha* estão conectados por um enlace também no estado *sem-falha* e o relógio de um deles é duas vezes mais rápido que o relógio do outro.

PROVA. De acordo com a especificação do algoritmo, após um nodo concluir o período de inicialização subsequente à sua recuperação, ele cria *tokens* e se torna um nodo testador naquele intervalo de testes. Se um nodo vizinho também conclui o período de inicialização após recuperar, ele também cria *tokens*, portanto ambos se tornam testadores simultâneos para seu enlace adjacente.

Se um nodo conclui o período de inicialização após recuperar enquanto um vizinho já está no estado *sem-falha*, este vizinho está testando o enlace correspondente uma vez a cada dois intervalos de testes, conforme provado no LEMA 1. Como o nodo que inicializa cria *tokens*, o vizinho no estado *sem-falha* pode também ser o testador daquele enlace naquele intervalo de testes.

Se ambos os nodos conectados por um enlace *não-respondendo* estão no estado *sem-falha*, então, de acordo com o LEMA 1, ambos estão testando aquele enlace a cada dois intervalos de testes. Como eles não estão trocando *tokens* entre si, ambos podem ser simultaneamente testadores quando seu enlace adjacente recupera.

Finalmente, considere o caso no qual o relógio de um nodo no estado *sem-falha* é duas vezes mais rápido que o relógio de um vizinho também no estado *sem-falha*, e ambos estão conectados por um enlace no estado *sem-falha*. Neste caso, tão logo o nodo mais rápido envia uma requisição de teste, ele reinicializa o temporizador do intervalo de testes para aquele enlace e se torna o nodo a ser testado. O novo testador, isto é, o nodo que recebeu a requisição de teste, também reinicializa o temporizador do intervalo de testes para aquele enlace. Uma vez que o temporizador do intervalo de testes deste nodo é duas vezes mais lento que o temporizador do intervalo de testes do outro nodo, o nodo mais rápido se torna o nodo testado e novamente o testador enquanto o nodo mais lento é ainda um testador. Portanto, periodicamente ambos os nodos se tornam testadores simultâneos.

□

TEOREMA 1. Quando dois nodos são testadores simultâneos, apenas um deles detém o *token* para o próximo intervalo de testes.

PROVA. Primeiramente considere que ambos os nodos são testadores simultâneos mas antes que um nodo envie uma requisição de teste, ele recebe uma requisição de teste, isto é, testes simultâneos não ocorrem. Aquele nodo, então, responde ao teste, e não testa o enlace naquele intervalo.

Agora considere que testes simultâneos ocorrem. De acordo com a especificação do algoritmo, quando um nodo executa um teste, ele atribui o valor TRUE à variável `TestRequestSent`, significando que, para aquele enlace, uma requisição de teste foi enviada. Se dois nodos executam testes simultâneos, então cada um deles terá sua variável `TestRequestSent` com o valor TRUE atribuído para o mesmo enlace. Como ambos os nodos recebem requisições de teste, ambos checam aquela variável e percebem que um teste simultâneo ocorreu. Neste caso, somente o nodo com o identificador menor responde, tornando-se o nodo testado. O outro nodo não responde ao teste e se comporta como o único testador naquele intervalo de testes. No intervalo de testes seguinte somente o atual nodo testado terá o *token*.

Uma situação notável ocorre quando dois nodos são testadores simultâneos e ambos enviam requisições de teste, mas a requisição de teste de um deles chega ao outro quando aquele nodo ainda não concluiu o tempo de espera para recuperação. De acordo com a DEFINIÇÃO 3, os testes enviados não são simultâneos, pois o teste de um dos nodos chega ao outro *antes* que aquele nodo envie a sua requisição de teste. Chamemos de nodo A o nodo cuja requisição de teste se perdeu e de nodo B o nodo que recebeu a requisição de teste durante o tempo de espera para recuperação. Se a requisição de teste enviada pelo nodo B chegar ao nodo A antes que aquele nodo detecte um *time-out*, então, de acordo com a especificação do algoritmo, o nodo A detecta a ocorrência de testes simultâneos por ter previamente atribuído o valor TRUE à variável `TestRequestSent[j]`.

Neste ponto, uma de duas situações pode ocorrer. Se o nodo A tiver o maior identificador, ele se tornará o nodo testador e também atribuirá TRUE à variável `TestRequestReceived[j]`.

Como sua requisição de teste se perdeu, aquele nodo detecta um *time-out* e, nesta ocasião, atribui o valor TRUE à variável `Token[j]`, tornando-se o testador para o intervalo de testes seguinte. Neste caso, o nodo B não recebe resposta ao teste que enviou; como não recebeu uma requisição de teste, também não detecta simultaneidade de testes. De acordo com a especificação do algoritmo, ele se torna o nodo testado para o próximo intervalo de testes. Portanto, permanece apenas um testador para o intervalo de teste seguinte.

Se, por outro lado, o nodo A tiver o menor identificador, ele assumirá o papel de nodo testado ao detectar simultaneidade de testes, enviando uma resposta e tomando o *token*, para ser o testador no próximo intervalo. O outro nodo, tendo realizado um teste com sucesso, torna-se o nodo testado. Portanto, em qualquer das duas situações (o nodo cuja requisição de testes se perdeu se torna o nodo testador ou o nodo testado ao detectar testes simultâneos), apenas um dos nodos terá o *token* para o intervalo de testes seguinte.

Desta forma, quando dois nodos são testadores simultâneos, apenas um deles detém o token para o intervalos de testes seguinte. □

LEMA 3. Após um nodo concluir o período de inicialização subsequente à sua recuperação e testar um enlace adjacente como *sem-falha*, o nodo testa aquele enlace uma vez a cada dois intervalos de testes, desde que o enlace permaneça no estado *sem-falha*.

PROVA. Um nodo que conclui o período de inicialização após recuperar testa um enlace adjacente como *sem-falha* tanto se o nodo vizinho acabou de concluir o período de inicialização após recuperar ou está no estado *sem-falha* e também o enlace está no estado *sem-falha*. De acordo com o LEMA 2, nestas situações ambos os nodos podem ser testadores simultâneos. O TEOREMA 1 assegura que somente um dos nodos detém o *token* para o próximo intervalo de testes, tanto se testes simultâneos são executados ou não. Além disso, se os nodos não são testadores simultâneos, o nodo que acabou de concluir o período de inicialização após uma recuperação envia uma requisição de teste ao outro com sucesso.

Em ambos os casos (os nodos se tornam simultaneamente testadores ou não), o nodo que envia uma requisição de teste também libera o *token*. O nodo que recebe a requisição de teste envia uma resposta e obtém o *token*, se tornando o testador para o

próximo intervalo de testes. A partir daquele intervalo, eles permanecem alternando seus papéis de testador e testado, desde que ambos os nodos e o enlace permaneçam no estado *sem-falha*. Portanto, o nodo que acabou de concluir o período de inicialização testa aquele enlace uma vez a cada dois intervalos de testes. □

LEMA 4. Se um nodo no estado *sem-falha* é adjacente a um enlace que se torna e permanece *não-respondendo*, o nodo testa aquele enlace uma vez a cada dois intervalos de testes.

PROVA. Um nodo executando DNR tanto testa como é testado através de determinado enlace em um dado intervalo. Se o enlace se torna *não-respondendo* e o nodo no estado *sem-falha* é o nodo testado no próximo intervalo de testes, ele não receberá uma requisição de teste naquele intervalo. Portanto, no próximo intervalo de testes o nodo atribui o valor TRUE à variável `TokenTurn` de forma que um *token* é criado no intervalo subsequente. Após o teste correspondente ser executado, o nodo libera o *token*, tornado-se o nodo a ser testado no intervalo de testes subsequente, como na situação inicial, e desta forma sucessivamente. Mas se o nodo é o testador após o enlace se tornar *não-respondendo*, ele libera o *token* tão logo ele executa um teste. Daquele intervalo de testes em diante, o mesmo padrão de comportamento descrito acima é repetido, uma vez que o nodo se torna o nodo testado e desde que o enlace se mantenha *não-respondendo*. Portanto o nodo se mantém testando o enlace uma vez a cada dois intervalos de teste. □

LEMA 5. Se um nodo está no estado *sem-falha* e um evento de *healing* ocorre em um enlace adjacente, este nodo se mantém testando aquele enlace uma vez a cada dois intervalos de testes desde que o enlace adjacente e os nodos se mantenham no estado *sem-falha*.

PROVA. Um evento de *healing* ocorre tanto porque um vizinho conclui o período de inicialização após recuperar como porque um enlace recupera enquanto ambos os nodos adjacentes estão no estado *sem-falha*. De acordo com o LEMA 2, quando um evento de *healing* ocorre, ambos os nodos podem se tornar simultaneamente testadores, e testes simultâneos podem ocorrer. O TEOREMA 1 garante que somente um nodo sucede como testador. Se os nodos não se tornam simultaneamente testadores, então ou o nodo ou seu

vizinho enviam uma requisição de teste ao outro.

Em ambos os casos (nodos se tornam simultanemente testadores ou não), o nodo que envia uma requisição de teste libera o *token*. O nodo que recebe a requisição de teste envia uma resposta e obtém o *token*, se tornando o testador para o próximo intervalo de testes. Daquele intervalo de testes em diante eles se mantém alternando seus papéis de testador e testado desde que ambos os nodos e o enlace se mantenham no estado *sem-falha*. Portanto o nodo que estava *sem-falha* e detectou o evento de *healing* testa aquele enlace uma vez a cada dois intervalos de testes. □

TEOREMA 2. Se o relógio de um nodo no estado *sem-falha* é mais do que duas vezes mais rápido que o relógio de um vizinho no estado *sem-falha* e ambos estão conectados por um enlace de comunicação também no estado *sem-falha*, então o único testador daquele enlace é o nodo mais rápido e ele testa o enlace uma vez a cada dois intervalos de testes.

PROVA. Tão logo o nodo mais rápido testa o enlace, o nodo vizinho responde ao teste e reinicializa o temporizador do intervalo de testes. O vizinho se torna o suposto testador do intervalo de testes seguinte. Entretanto, o intervalo de testes do nodo mais rápido expira duas vezes antes do instante de tempo no qual se supõe que o vizinho execute um teste. Portanto o próximo teste é executado novamente pelo nodo mais rápido.

Em outras palavras: após o temporizador do intervalo de testes do nodo mais rápido expirar pela primeira vez, aquele nodo atribui o valor TRUE à variável `TokenTurn`. Quando o intervalo de testes expira novamente, o nodo mais rápido executa um teste, e isto ocorre antes que o intervalo de testes do nodo vizinho expire. Após este novo teste ser executado, o outro nodo reinicializa seu intervalo de testes novamente, e desta forma sucessivamente.

Portanto, neste caso, somente o nodo mais rápido testa o enlace a cada dois intervalos de testes. □

TEOREMA 3. Se o relógio de um nodo no estado *sem-falha* é exatamente duas vezes mais rápido que o relógio de um vizinho no estado *sem-falha* e eles estão conectados por um enlace de comunicação também no estado *sem-falha*, então ou eles se alternam como testador e testado ou apenas um deles testa o enlace, dependendo de seus identificadores.

PROVA. De acordo com o LEMA 2 neste caso ambos os nodos se tornam simultaneamente testadores periodicamente. Se o nodo mais rápido também tém o maior identificador, a cada vez que os testes simultâneos ocorrem, aquele nodo se torna o testador. Assumindo que testes simultâneos sempre ocorrem, então o enlace é testado uma vez a cada dois intervalos de testes pelo nodo com o maior identificador. Por outro lado, se o nodo mais rápido tém o menor identificador, a cada vez que os testes simultâneos ocorrem, ele se torna o nodo testado. Em seu próximo intervalo de testes, o nodo mais rápido executa um teste. Dois intervalos de teste depois disso, testes simultâneos ocorrem de novo, e desta forma sucessivamente. □

TEOREMA 4. Um nodo no estado *sem-falha* executando DNR testa um enlace adjacente uma vez a cada dois intervalos de testes, a menos que seu relógio seja duas vezes ou mais de duas vezes mais rápido que o relógio de um vizinho também no estado *sem-falha*.

PROVA. Se o relógio de um nodo no estado *sem-falha* é mais de duas vezes mais rápido que o relógio de um vizinho no estado *sem-falha*, e eles estão conectados por um enlace de comunicação também no estado *sem-falha*, então o nodo mais rápido é o único testador daquele enlace, conforme demonstrado no TEOREMA 2. Portanto, nesta situação, o nodo mais lento nunca testa o enlace. Se o relógio de um nodo no estado *sem-falha* é exatamente duas vezes mais rápido que o relógio de um vizinho no estado *sem-falha*, e eles estão conectados por um enlace de comunicação também no estado *sem-falha*, então ou eles alternam-se como nodos testador e testado após a ocorrência de testes simultâneos ou somente o nodo mais rápido testa o enlace, conforme demonstrado no TEOREMA 3.

Por outro lado, se a velocidade do relógio de um nodo é menos de duas vezes mais lenta que a do seu vizinho, então o nodo testa seu enlace adjacente uma vez a cada dois intervalos de testes. Isto procede do conjunto de LEMAS acima. Se um nodo recupera e detecta um enlace adjacente como *não-respondendo*, o LEMA 1 prova que o nodo testa aquele enlace uma vez a cada dois intervalos de testes, desde que o enlace permaneça no estado *não-respondendo*. Se um nodo recupera e detecta um enlace adjacente como *sem-falha*, o LEMA 3 prova que o nodo testa aquele enlace uma vez a cada dois intervalos

de testes, desde que o enlace permaneça no estado *sem-falha*. Se um nodo está no estado *sem-falha* e um enlace adjacente no estado *sem-falha* se torna *não-respondendo*, então o nodo continua testando aquele enlace uma vez a cada dois intervalos de testes conforme demonstrado pelo LEMA 4. Finalmente, se um nodo está no estado *sem-falha* e um evento de *healing* ocorre em um enlace adjacente previamente no estado *não-respondendo*, o nodo continua testando aquele enlace uma vez a cada dois intervalos de testes, conforme demonstrado pelo LEMA 5.

Portanto, como estes são os únicos casos possíveis, um nodo no estado *sem-falha* executando DNR testa um enlace adjacente uma vez a cada dois intervalos de testes, a menos que a velocidade de seu relógio seja duas vezes ou mais de duas vezes mais lenta que a velocidade de relógio de um nodo vizinho no estado *sem-falha*. □

Para todas as provas abaixo assume-se que a diferença das velocidades dos relógios de quaisquer dois nodos vizinhos é sempre menos que o dobro uma da outra.

COROLÁRIO 1. Se ambos os nodos conectados por um enlace no estado *sem-falha* permanecem no estado *sem-falha* por mais de um intervalo de testes, então somente um teste é executado naquele enlace por intervalo de testes, desde que os nodos e os enlaces permaneçam naquele estado.

PROVA. Este COROLÁRIO procede do TEOREMA 4: como cada nodo no estado *sem-falha* executando DNR testa um enlace adjacente uma vez a cada dois intervalos de testes, se o enlace permanece no estado *sem-falha*, os nodos devem alternar-se em seus papéis de testador e testado em intervalos sucessivos, de forma que somente um teste é executado por enlace por intervalo de testes. □

COROLÁRIO 2. Se um nodo é o único nodo no estado *sem-falha* adjacente a um enlace no estado *não-respondendo* por mais de um intervalo de testes, então somente um teste é executado naquele enlace a cada dois intervalos de testes enquanto o enlace se mantiver naquele estado.

PROVA. Este COROLÁRIO também procede do TEOREMA 4: como cada nodo no estado *sem-falha* executando DNR testa um enlace adjacente uma vez a cada dois intervalos de testes, se um enlace se mantém no estado *não-respondendo* com somente um nodo

adjacente no estado *sem-falha*, então aquele nodo se torna testador uma vez a cada dois intervalos de testes. □

Em seguida é provado o pior caso de latência de detecção de eventos da estratégia proposta.

LEMA 6. O estado *não-respondendo* de um enlace é detectado em no máximo dois intervalos de testes.

PROVA. O estado *não-respondendo* de um enlace é detectado quando um dos nodos adjacentes se recupera ou se um dos nodos está no estado *sem-falha* enquanto o enlace se torna *não-respondendo*. Se um nodo está recuperando, o estado de *não-respondendo* é detectado tão logo ocorra um *time-out* da resposta esperada após o nodo enviar requisições de teste para todos os seus vizinhos.

Se um nodo está no estado *sem-falha* enquanto um enlace se torna *não-respondendo*, aquele nodo pode ser tanto o testador como o nodo testado naquele intervalo de testes. Se o nodo é o testador, ele detecta o estado de *não-respondendo* em no máximo um intervalo de testes, tão logo ele teste o enlace. Por outro lado, se o nodo no estado *sem-falha* é o nodo a ser testado naquele intervalo de testes, aquele nodo vai-se tornar o testador no intervalo seguinte, conforme demonstrado no LEMA 4, portanto ele detecta o estado *não-respondendo* do enlace em no máximo dois intervalos de testes. □

Deve-se notar que provamos acima o pior caso de latência: informação sobre um novo evento pode ser obtida antes, até mesmo com uma detecção ocorrida durante uma disseminação.

LEMA 7. Um evento de *healing* é detectado em no máximo dois intervalos de testes.

PROVA. O evento de *healing* de um enlace é detectado por um nodo no estado *sem-falha* se um nodo adjacente conclui o período de inicialização após recuperar-se enquanto o enlace está recuperando ou está no estado *sem-falha*, ou se um vizinho está no estado *sem-falha*, enquanto o enlace previamente no estado *não-respondendo* recupera.

Se um nodo recupera, após o período de inicialização ele envia requisições de teste a todos os seus vizinhos. Neste caso, se o enlace adjacente está recuperando ou está no estado *sem-falha*, então devido aos testes *two-way*, tão logo o vizinho no estado *sem-*

falha responde, o estado do testador é detectado como *sem-falha*, da mesma forma que é detectado como *sem-falha* o estado do seu enlace de comunicação.

Se ambos os nodos adjacentes estavam no estado *sem-falha* enquanto o enlace recuperou, aqueles nodos não estavam alternando-se em seus papéis de testador e testado previamente ao evento de *healing*. Portanto, tanto o enlace pode como pode não ter um testador para aquele intervalo de testes, porque ambos os nodos podem ser nodos a serem testados naquele intervalo. Neste caso, o enlace vai ser testado somente no intervalo de testes subsequente, conforme demonstrado no LEMA 4. Devido à estratégia *two-way* de testes, o nodo testado também detecta o evento de *healing*. □

Em ambos os casos acima, se testes simultâneos ocorrem, eles não afetam a latência de detecção de eventos, uma vez que somente um testador permanece, mas existe também a estratégia *two-way*.

TEOREMA 5. A latência de detecção de eventos do DNR é de dois intervalos de testes no pior caso.

PROVA. Primeiramente considere um nodo executando DNR o qual está recuperando ou está no estado *sem-falha*. Este nodo detecta o estado *não-respondendo* de um enlace tanto porque um nodo adjacente no estado *sem-falha* e/ou seu enlace falham. A latência de detecção destes casos é no máximo dois intervalos de testes, conforme demonstrado pelo LEMA 6.

O evento de *healing* de um enlace é detectado por um nodo no estado *sem-falha* se um nodo adjacente recupera e se o enlace adjacente está no estado *sem-falha* ou se um enlace recupera enquanto o nodo adjacente está recuperando ou está no estado *sem-falha*. A latência de detecção de tais eventos é também de no máximo dois intervalos de testes, conforme demonstrado pelo LEMA 7.

Como estes são os únicos casos possíveis, a latência de detecção de eventos do DNR é de, no máximo, dois intervalos de testes. □

3.3.2 Provas da Fase de Disseminação

Esta Seção prova tanto a latência de disseminação de informações após a detecção de um evento como a correção do procedimento de *healing*. Inicialmente demostra-se como a ocorrência de eventos durante a disseminação pode afetar sua latência.

A latência de disseminação de informações sobre eventos no DNR é computada considerando o início de uma mensagem de disseminação após a detecção de um evento. Após a detecção de um evento em um enlace adjacente, um nodo executando DNR desencadeia a disseminação de informações sobre o evento utilizando uma estratégia paralela. Desta forma, quando uma disseminação é iniciada, uma mensagem é enviada através de todos os seus enlaces adjacentes, atingindo os vizinhos sem-falha, os quais a reencaminham pela rede de maneira análoga.

Portanto, uma disseminação pode seguir por múltiplos caminhos. Seja uma mensagem *redundante* quando ela chega em um nodo por onde ela já foi disseminada. Seja o *caminho de uma disseminação* o conjunto de enlaces sobre os quais a mensagem é subsequentemente encaminhada de tal forma que chega a cada nodo antes de uma mensagem redundante. Seja um *nível* o conjunto de nodos que estão *no caminho daquela disseminação* com a mesma distância do nodo que desencadeou a disseminação.

DEFINIÇÃO 4. Uma *rodada de disseminação* é definida como o intervalo de tempo no qual todos os nodos em um nível do caminho da disseminação enviam suas mensagens para todos os nodos no nível seguinte.

Seja o diâmetro de um componente conexo a maior distância mínima entre dois nodos quaisquer naquele componente. A latência de disseminação em um componente conexo é diretamente proporcional ao seu diâmetro. É importante observar que o diâmetro pode mudar com a ocorrência de novos eventos.

Diz-se que um evento altera o diâmetro de um componente conexo *para uma determinada disseminação* quando o evento ocorre em um nodo ou enlace ainda não alcançado pela disseminação naquele componente conexo.

Um evento de falha pode refletir tanto a falha de um enlace ou a falha de um nodo. Se o evento não for detectado até o momento em que a disseminação alcança

aquele elemento da rede, é detectado pela própria disseminação. Um evento de *healing*, por outro lado, somente é detectado pela chegada de uma requisição de teste, embora um enlace recuperado mas ainda não detectado como tal possa ser utilizado por uma disseminação. Se o evento de *healing* é causado pela recuperação de um nodo, então ou o enlace vai ser utilizado quando a disseminação chega àquele nodo ou a informação vai ser encaminhada pelo próprio procedimento de *healing*.

TEOREMA 6. Considere que uma disseminação é desencadeada após a detecção de um evento e prossegue por um componente conexo onde outros eventos podem ocorrer. Considere a sucessão dos eventos que alteram o diâmetro daquele componente conexo para aquela disseminação. Seja d o diâmetro do componente após o último de tais eventos ocorrer. São necessárias no máximo d rodadas de disseminação desde o início da disseminação até que ela chegue a todos os nodos no componente conexo final.

PROVA. Como uma disseminação é propagada com uma estratégia paralela, em um componente conexo com diâmetro d, o caminho mais longo tém tamanho no máximo igual d. Portanto, de acordo com a definição de uma rodada de disseminação acima, são necessárias no máximo d rodadas de disseminação para que a mensagem sendo disseminada chegue a todos os nodos naquele componente. □

A seguir é provada a correção do procedimento de *healing*.

A mensagem enviada pelo nodo testado ao nodo testador após a detecção de um evento de *healing* é chamada de *mensagem de healing*. A mensagem de healing contém um extrato da tabela de estados com todos os enlaces cujos *timestamps* são maiores do que 1.

O envio de uma mensagem de *healing* e o subsequente envio de uma mensagem de disseminação pelo nodo que recebe a mensagem de *healing* é chamado de um *procedimento de healing*.

TEOREMA 7. O *procedimento de healing* assegura que informação completa sobre os componentes prévios ao *healing* chegue a todo o novo componente conexo formado.

PROVA. Para o procedimento de *healing* assume-se que os nodos que detectaram o evento de *healing* possuem informação atualizada sobre todos os eventos ocorridos em seus

componentes conexos previamente ao evento de *healing*. Com efeito, se algum deles não possui informação completa sobre estes eventos, isto é devido ao fato de que há eventos ainda sendo disseminados. Estes eventos alcançarão no futuro o enlace que sofreu *healing* e serão por sua vez disseminados através deste enlace, chegando, portanto, a todo o novo componente conexo, desde que novos eventos não ocorram.

Como aos *timestamps* de enlaces *inatingíveis* foi atribuído o valor 1, o menor *timestamp* possível, nas tabelas de enlaces locais de todos os nodos nos componentes conexos prévios, a mensagem de *healing* gerada contém somente informações sobre enlaces localizados no componente original do nodo que a envia. Como a mensagem de disseminação gerada pelo nodo que recebe a mensagem de *healing*, por sua vez, é gerada após a atualização da tabela de enlaces local daquele nodo, e após o incremento do *timestamp* do enlace que sofreu *healing*, aquela mensagem contém informação sobre ambos os componentes conexos anteriores ao *healing* mais informação sobre o próprio evento de *healing*.

Assim, embora parte da informação contida na mensagem de disseminação final gerada durante o procedimento de *healing* seja redundante em diferentes porções da rede, ela contém informação completa sobre o componente conexo recém formado. □

CAPÍTULO 4

PROVA DE CORREÇÃO NO ARCABOUÇO *BOUNDED CORRECTNESS*

Bounded Correctness [Subbiah e Blough, 2004] é um modelo formal do comportamento dinâmico para algoritmos de diagnóstico em nível de sistema. O modelo é apresentado como um conjunto de propriedades que definem o que significa a correção e o desempenho de algoritmos de diagnóstico na presença de falhas e recuperações dinâmicas. Um *evento* é definido como uma transição de estado, ocorrida em uma unidade diagnosticável do sistema. De forma a ser *bounded correct*, todos os nodos sem-falha executando um algoritmo de diagnóstico precisam registrar a ocorrência de eventos tão rapidamente quanto possível. Nodos que inicializam devem obter uma visão dos estados das demais unidades do sistema dentro de um limite de tempo. Além disso, o algoritmo tem que garantir que os nodos sem-falha não detectam eventos espúrios.

Este Capítulo está organizado como segue: inicialmente são apresentadas algumas definições bem como as propriedades que caracterizam *bounded correctness*. Em seguida, o algoritmo DNR é avaliado segundo este modelo. São obtidas as latências de diagnóstico do algoritmo, seus *tempos de retenção de estado* e a latência de inicialização. Com base nestes resultados, prova-se que o algoritmo DNR satisfaz as propriedades de *bounded correctness*, sendo, portanto, correto dentro deste arcabouço.

4.1 DEFINIÇÕES INICIAIS

Nesta Seção serão apresentadas as definições básicas do arcabouço de *bounded correctness*. Antes de citar as propriedades do modelo, será generalizada a definição de um estado *T-válido* de forma a incluir enlaces do sistema como segue.

definição 5. *Um estado mantido por um nodo sem-falha X para outro nodo Y ou enlace Y–Z no instante de tempo t é dito* **T-válido** *se aquele nodo ou enlace esteve no*

estado indicado em algum ponto durante o intervalo de tempo $[t - T, t]$.

PROPRIEDADE 1. **Bounded Diagnostic Latency** ou, em português, **Latência Delimitada de Diagnóstico**. *Considere um evento ocorrido no sistema num instante de tempo arbitrário t. Qualquer nodo que esteja no estado sem-falha continuamente no intervalo* $[t, t + L]$ *deve ter diagnosticado o evento até o tempo* $t + L$*, onde L é um limite de tempo próprio do algoritmo referido como sua* latência de diagnóstico.

PROPRIEDADE 2. **Bounded Start-Up** ou, em português, **Latência Delimitada de Inicialização**. *Considere um nodo arbitrário X recuperando no tempo t. Se X permanecer no estado sem-falha continuamente durante o intervalo de tempo* $[t, t + S]$*, então no tempo* $t + S$*, X deve ter estados L-válidos para todos os nodos e enlaces do sistema, onde* $S \geq L$ *é um intervalo de tempo chamado de* latência de inicialização.

PROPRIEDADE 3. **Accuracy** ou, em português, **Acuidade**. *Considere um nodo arbitrário sem-falha X, após ter inicializado. Cada transição de estado mantida por X para qualquer outro nodo ou enlace do sistema deve corresponder a um evento efetivamente ocorrido naquela unidade bem como a ocorrência de um evento não deve causar transições de estado múltiplas em X.*

De forma a provar que o algoritmo DNR é *bounded correct*, deve-se considerar todos os tipos de eventos que podem possivelmente ocorrer tanto em nodos como em enlaces; além disso, partições e reconexões da rede também são possíveis. As definições 6 a 11 são originalmente de [SUBBIAH e BLOUGH, 2004]; as definições 8, 12 e 13 foram generalizadas neste trabalho.

DEFINIÇÃO 6. *O* tempo de inicialização da comunicação*, do inglês* send initiation time*,* Δ_{send_init}*, é o intervalo de tempo desde o instante em que um nodo inicia uma comunicação até que o último* bit *de uma mensagem seja injetado na rede.*

Como em [SUBBIAH e BLOUGH, 2004], assume-se que Δ_{send_init} é uma constante.

DEFINIÇÃO 7. *Os* atrasos mínimos *e* máximos na rede*, do inglês* minimum *and* maximum message delays*,* Δ_{send_min} *e* Δ_{send_max}*, são os intervalos de tempo mínimo e máximo, respectivamente, desde o instante em que o último* bit *de uma mensagem é injetado na rede até que a mensagem seja completamente entregue a um nodo vizinho*

sem-falha.

DEFINIÇÃO 8. *O* tempo de retenção de estado*, do inglês* state holding time *é o intervalo de tempo mínimo que um nodo ou enlace deve se manter em um estado antes de fazer uma transição para outro estado.*

DEFINIÇÃO 9. Tempo real *é o tempo físico Newtoniano.*

DEFINIÇÃO 10. Tempo de relógio*, do inglês* clock time *é o tempo diretamente observável no relógio de um nodo. O* tempo de relógio *de um nodo X no tempo real t é denotado por* $T_X(t)$.

DEFINIÇÃO 11. *O relógio de um nodo no estado sem-falha experimenta uma taxa de oscilação limitada (*bounded drift*). Se um nodo X se mantém no estado sem-falha continuamente durante o intervalo* $[t_1, t_2]$*, então para todos os intervalos de tempo reais* $[u_1, u_2] \subseteq [t_1, t_2]$, $\mid [T_x(u_2) - T_x(u_1)] - (u_2 - u_1) \mid \leq \rho(u_2 - u_1)$*, onde* $\rho \ll 1$ *é a taxa máxima de oscilação de um relógio.*

DEFINIÇÃO 12. *Considere um nodo arbitrário X que reinicializa o* timer *do intervalo de testes no instante de tempo t e se mantém no estado sem-falha indefinidamente. O nodo X vai reinicializar novamente o* timer *do intervalo de testes não após o tempo real* $t + (1 + \rho)\pi$*, onde* π *é o intervalo de testes.*

DEFINIÇÃO 13. *O* tempo de espera para recuperação*, do inglês* recovery wait time*,* W*, é o tempo local de relógio pelo qual um nodo que acabou de entrar no estado sem-falha espera antes de tomar parte em uma rodada de testes.*

O *tempo de espera para recuperação* é útil para permitir que nodos se mantenham falhos por períodos curtos de tempo e mesmo assim garantam que seus vizinhos detectem aquele evento.

4.2 *BOUNDED CORRECTNESS* DO ALGORITMO DNR

De forma a provar que o algoritmo DNR é *bounded correct*, a latência de diagnóstico do algoritmo é avaliada segundo aquele modelo. Resultados de *tempos de retenção de estado* são então obtidos, da mesma forma que a latência de inicialização do algoritmo. Finalmente, as propriedades (*latência delimitada de diagnóstico*, *latência delimitada de*

inicialização e *acuidade*) são provadas.

A análise abaixo procede se $\pi > \Delta_{send\_max} > \Delta_{send\_min} > \Delta_{send\_init}$, o que efetivamente ocorre em redes reais. Como em [SUBBIAH e BLOUGH, 2004], os termos em ρ^2 são ignorados nas provas.

4.2.1 Latência de Diagnóstico

DEFINIÇÃO 14. A *latência de diagnóstico* do algoritmo DNR é o tempo máximo requerido por um nodo para detectar um evento em um enlace adjacente (chamada latência de detecção de evento) mais o tempo máximo para disseminar a informação sobre o novo evento no componente conexo ao qual o nodo pertence (chamada latência de disseminação) mais Θ, o tempo máximo requerido para computar a alcançabilidade da rede.

Como a alcançabilidade da rede pode ser computada com uma busca em largura no grafo, sua complexidade é $O(N)$, onde N é o número total de nodos na rede.

LEMA 8. A latência máxima de disseminação do DNR é $D(\Delta_{send\_init} + \Delta_{send\_max})$, onde D é o maior diâmetro possível da rede dada qualquer configuração de falha.

PROVA. Como nodos executando DNR disseminam informação de forma paralela, a latência de disseminação é proporcional no diâmetro da rede durante o tempo de disseminação. Seja D o maior possível diâmetro. Como uma mensagem de disseminação leva no máximo $\Delta_{send\_init} + \Delta_{send\_max}$ para atravessar um enlace, o LEMA está provado. □

No lema 9 abaixo um *time-out* de teste corresponde ao intervalo de tempo máximo que um nodo leva para detectar a ausência de resposta a um teste após uma requisição de teste ser enviada, isto é, para detectar o enlace como *não-respondendo*.

Muitas estratégias podem ser usadas para computar este *time-out* tão precisamente quanto possível, por exemplo os intervalos crescentes propostos por [LARREA, FERNÁNDEZ e ARÉVALO, 2
ou o algoritmo utilizado pelo protocolo TCP (*Transmission Control Protocol*) da Internet, proporcional ao *round-trip time* e em seu desvio médio [POSTEL, 1981].

LEMA 9. Se o *round-trip time* para a troca de mensagens entre vizinhos é $2(\Delta_{send\_init} + \Delta_{send\_max})$, o *time-out* empregado por um nodo executando DNR é $2(1 + 2\rho)(\Delta_{send\_init} + \Delta_{send\_max})$.

PROVA. O tempo máximo necessário para uma requisição de teste chegar a um nodo testado é $\Delta_{send\_init} + \Delta_{send\_max}$. O máximo que a resposta correspondente leva para chegar ao nodo testador é também $\Delta_{send\_init} + \Delta_{send\_max}$. De forma a lidar com uma oscilação máxima possível de $(1 - \rho)$, no tempo em que o testador envia a requisição de teste, ele inicializa o *timer* do *time-out* de teste para $2(1 + \rho)(\Delta_{send\_init} + \Delta_{send\_max})$. Então se a oscilação do *timer* é $(1 + \rho)$, o intervalo correspondente de tempo é no máximo $2(1 + 2\rho)(\Delta_{send\_init} + \Delta_{send\_max})$ se termos em ρ^2 são ignorados. □

Os lemas abaixo provam os limites de latência do algoritmo DNR. Primeiramente a latência de diagnóstico de um evento em que um enlace se torna *não-respondendo* é avaliada. Para um enlace tornar-se *não-respondendo* no DNR, tanto um nodo adjacente pode se tornar falho como o próprio enlace pode ter falhado. O nodo falho adjacente pode tanto ter assumido o papel de testador como de testado para o intervalo de teste seguinte. No caso de um enlace ter falhado, ele pode ter particionado a rede ou não. Cada um destes casos têm latências de diagnóstico específicas. Além disso, para calcular a latência máxima, considera-se que um nodo pode assumir o papel de nodo testado para o intervalo de teste seguinte tão logo ele *conclui* uma requisição de teste. De maneira similar, um nodo se torna o testador tão logo ele recebe uma requisição de teste e envia a respectiva resposta. No caso da falha de enlaces, a maior latência possível de diagnóstico ocorre tão logo uma requisição de teste e a respectiva resposta são completamente enviados através do enlace.

LEMA 10. A latência de diagnóstico de um evento em que um enlace se torna *não-respondendo* é de no máximo $2(1 + \rho)\pi + (D + 4\rho)\Delta_{send\_init} + (D + 2 + 4\rho)\Delta_{send\_max} - \Delta_{send\_min} + \Theta$.

PROVA. Inicialmente considere a falha de um enlace que não particiona a rede. O pior caso de latência ocorre se um nodo Y envia uma requisição de teste para o nodo X no tempo t. O nodo Y recebe uma resposta de teste o mais cedo possível no tempo $t + 2(\Delta_{send\_init} + \Delta_{send\_min})$ e então o enlace falha. No instante de tempo em que o nodo X envia uma resposta de teste ele também reinicializa seu *timer* do intervalo de testes. Então o nodo X envia uma requisição de teste no tempo $t + \Delta_{send\_init} + \Delta_{send\_min} + (1 + \rho)\pi$

e detecta o evento quando o *time-out* expira. Subtraindo o instante de tempo em que o enlace falhou do instante de tempo mais tardio possível de detecção e adicionando o tempo de disseminação mais Θ resulta $(1+\rho)\pi+(D+1+4\rho)\Delta_{send\_init}+(D+2+4\rho)\Delta_{send\_max}-\Delta_{send\_min}+\Theta$ como latência de diagnóstico para este caso.

Agora o caso em que um enlace particiona a rede é considerado. Da mesma forma que no caso acima, suponha que um nodo X envia uma requisição de teste para o nodo Y no tempo t e receba uma resposta no instante de tempo $t+2(\Delta_{send\_init}+\Delta_{send\_min})$, imediatamente antes de o enlace falhar. O nodo X não recebe a próxima requisição de teste, enviada por Y, no tempo $t+\Delta_{send\_init}+\Delta_{send\_min}+(1+\rho)\pi$, quando aquele nodo detecta que o enlace se tornou *não-respondendo*. No tempo $t+(1+\rho)\pi$, o nodo X atribui TRUE à variável `TokenTurn` e envia uma requisição de teste no tempo $t+2(1+\rho)\pi$. O *time-out* de X expira em $t+2(1+\rho)\pi+2(1+2\rho)(\Delta_{send\_init}+\Delta_{send\_max})$. Subtraindo o instante de tempo em que o enlace falhou do instante mais tarde em que o evento pode ser detectado resulta na latência de detecção. Adicionando o tempo de disseminação mais Θ resulta $2(1+\rho)\pi+(D+4\rho)\Delta_{send\_init}+(D+2+4\rho)\Delta_{send\_max}-2\Delta_{send\_min}+\Theta$ como latência de diagnóstico.

A partir deste ponto é avaliada a latência de diagnóstico de um evento em que um enlace se torna *não-respondendo* quando o nodo adjacente falha após ter assumido o papel de nodo testado para o intervalo de testes seguinte. Suponha que o nodo Y falhe no tempo $t+\Delta_{send\_init}$, imediatamente após ter enviado uma requisição de teste. O node X recebe a requisição de teste no máximo no tempo $t+\Delta_{send\_init}+\Delta_{send\_max}$ e envia uma resposta. O nodo X envia uma requisição de teste no máximo no tempo $t+\Delta_{send\_init}+\Delta_{send\_max}+(1+\rho)\pi$ e o *time-out* expira após $2(1+2\rho)(\Delta_{send\_init}+\Delta_{send\_max})$. Subtraindo o tempo em que o nodo falhou do instante mais tardio em que ocorre a detecção e adicionando o tempo de disseminação mais Θ resulta em uma latência de diagnóstico igual a $(1+\rho)\pi+(D+2+4\rho)\Delta_{send\_init}+(D+3+4\rho)\Delta_{send\_max}+\Theta$.

Agora suponha que o nodo X envia uma requisição de teste no tempo t. O nodo Y recebe a requisição de teste o mais cedo possível no tempo $t+\Delta_{send\_init}+\Delta_{send\_min}$ e falha após Δ_{send_init}, imediatamente após ter enviado uma resposta. O nodo Y assumiu

o papel de testador. O nodo X recebe uma resposta no máximo após Δ_{send_max}. O nodo X atribui TRUE à variável `TokenTurn` no tempo $t + (1 + \rho)\pi$ e envia uma requisição de teste no tempo $t + 2(1 + \rho)\pi$. Após o *time-out*, o nodo X detecta o estado de *não-respondendo* do enlace. Subtraindo o tempo em que o nodo falhou do tempo mais tardio possível de detecção e adicionando o tempo de disseminação mais Θ resulta na latência de diagnóstico. Este é o pior caso de latência para detectar um evento em que um enlace se torna *não-respondendo*, e o LEMA segue. □

LEMA 11. A latência de diagnóstico de um evento de *healing* é de no máximo $2(1 + \rho)\pi + (D + 1)\Delta_{send\_init} + (D + 2)\Delta_{send\_max} - \Delta_{send\_min} + \Theta$.

PROVA. Eventos de *healing* no DNR ocorrem tanto quando um nodo inicializa pela primeira vez ou quando ele recupera, ou ainda quando um enlace falho incidente em dois nodos sem-falha recupera. Primeiramente considere a latência de diagnóstico de um evento de *healing* quando um nodo recupera. Após recuperar no tempo t, um nodo aguarda o *tempo de espera para recuperação* W antes de enviar requisições de teste aos seus vizinhos. Em tempo real, isto pode levar no máximo $(1+\rho)W$. Os nodos vizinhos recebem as requisições de teste no máximo após um tempo de $\Delta_{send\_init} + \Delta_{send\_max}$ e detectam que seu enlace adjacente recuperou. Os vizinhos então respondem com informação sobre a tabela de enlaces. O nodo em recuperação recebe estas mensagens no máximo após $\Delta_{send\_init} + \Delta_{send\_max}$. Com o subsequente início de uma mensagem de disseminação pelo nodo que recuperou, a latência de diagnóstico resulta em, no máximo, $(1 + \rho)W + (D + 2)(\Delta_{send\_init} + \Delta_{send\_max}) + \Theta$. Neste trabalho W é avaliado de forma a ser minimizado, sendo da ordem de $\pi/2$. Portanto a latência de diagnóstico para este caso de *healing* é menor que o pior caso de latência enunciado no lema.

Quando testes simultâneos ocorrem, se a requisição de teste do nodo com o maior identificador chega ao outro nodo antes que aquele nodo complete o *tempo de espera para recuperação*, então o nodo testador vai testar novamente após um intervalo de testes. Neste caso, o evento de *healing* será diagnosticado no máximo no tempo $t + (1 + \rho)W + (1 + \rho)\pi + (D + 2)(\Delta_{send\_init} + \Delta_{send\_max}) + \Theta$. Novamente, este não é o pior caso de latência enunciado no LEMA.

O pior caso de latência enunciado neste LEMA ocorre quando um enlace falho recupera enquanto ambos os nodos estão sem-falha e testando aproximadamente ao mesmo tempo. Suponha que um enlace falho recupere no tempo $t + \Delta_{send\_init} + \Delta_{send\_min}$ imediatamente após o nodo X ter enviado uma requisição de teste. Suponha que o nodo Y teste após t, mas antes que o enlace recupere. Portanto, o nodo X envia uma requisição de teste novamente no tempo $2(1 + \rho)\pi$. O nodo Y recebe a requisição de teste no máximo após $\Delta_{send\_init} + \Delta_{send\_max}$ e detecta que o enlace *não-respondendo* recuperou. O nodo Y envia uma resposta, que o nodo X recebe no máximo no tempo $t+2(1+\rho)\pi+2(\Delta_{send\_init}+\Delta_{send\_max})$, antes que seu *timeout* de teste expire. Isto resulta em uma latência de detecção de $2(1+\rho)\pi+\Delta_{send\_init}+2\Delta_{send\_max}-\Delta_{send\_min}$. No tempo em que o nodo X detecta o evento de *healing*, ele inicia a disseminação de informação de diagnóstico e o LEMA segue. □

COROLÁRIO 3. Para o pior caso de latência de detecção de um evento de *healing* ocorrer, a maior diferença de tempo entre o envio de requisições de teste por ambos os nodos adjacentes é Δ_{send_min}.

PROVA. No caso descrito no LEMA 11, o enlace recupera no tempo $t' = t + \Delta_{send\_init} + \Delta_{send\_min}$ após o nodo Y enviar uma requisição de teste. Se o nodo X enviar uma requisição de teste no tempo Δ_{send_init} antes de t', o teste é injetado na rede exatamente no tempo em que o enlace recupera, o nodo Y recebe a requisição de teste e o evento é detectado imediatamente. Se a requisição de teste é enviada antes daquele tempo, ela é perdida devido à falha do enlace, e a recuperação vai ser detectada somente após dois intervalos de teste adicionais. A diferença entre os tempos em que cada nodo enviou uma requisição de teste é Δ_{send_min}. Portanto, esta é a máxima diferença de tempo permitida para que ambos os testadores enviem requisições de teste de forma que o pior caso de latência de detecção de um evento de *healing* ocorra. □

4.2.2 Tempos de Retenção de Estado

Os lemas abaixo calculam os tempos de retenção de estado para o algoritmo.

LEMA 12. O tempo mínimo que um nodo deve permanecer no estado *sem-falha* de forma a ter este estado diagnosticado é $(1+\rho)W + (1+\rho)\pi + (D+2)\Delta_{send\_init} + (D+2)\Delta_{send\_max} + \Theta$.

PROVA. Como um nodo envia requisições de teste a todos os seus vizinhos após recuperar, o tempo mínimo que um nodo deve permanecer neste novo estado, isto é, o *tempo de retenção de estado para nodos no estado sem-falha*, do inglês *node working state holding time*, SHT_{w_node}, é computado como segue: o *tempo de espera para recuperação* W, que é calculado abaixo, é somado com o intervalo de tempo máximo necessário para enviar requisições de teste e receber informações das tabelas de estados como resposta, o qual é $2(\Delta_{send\_init} + \Delta_{send\_max})$, mais o tempo para disseminar as mensagens de *healing* finais por toda a rede. Se, por outro lado, testes simultâneos ocorrem, o processo de *healing* pode levar ainda um intervalo de testes extra para concluir, e o lema segue. □

LEMA 13. O tempo mínimo que um enlace deve permanecer no estado *sem-falha* de forma a ter este estado diagnosticado é $2(1+\rho)\pi + (D+1)\Delta_{send\_init} + (D+2)\Delta_{send\_max} - \Delta_{send\_min} + \Theta$.

PROVA. Como um enlace não tém *tempo de espera para recuperação* tampouco envia requisições de teste, o *tempo de retenção de estado para enlaces no estado sem-falha*, do inglês *link working state holding time*, SHT_{w_link}, deve ser maior que a latência máxima de diagnóstico calculada para o diagnóstico de enlaces recentemente recuperados, no LEMA 11. □

O tempo de retenção de estado acima é válido desde que ambos os nodos adjacentes ao enlace estejam no estado *sem-falha*. De fato, se um dos nodos está falho, o enlace é ainda considerado *não-respondendo*, independentemente de quanto tempo ele permanece sem-falha. Se ambos os nodos estão falhos, o enlace é considerado *inatingível* após ambos os nodos serem também diagnosticados como *inatingíveis*. A despeito disto, o enlace é considerado *sem-falha* tão logo ambos os nodos recuperem e enviem requisições de teste.

LEMA 14. O tempo mínimo que um nodo deve permanecer falho de forma a ter diagnosticado o estado *não-respondendo* de todos os seus enlaces adjacentes é $(3+4\rho)\pi/2 + (2D+1+\rho)\Delta_{send\_init}/2 + (2D+5(1+\rho))\Delta_{send\_max}/2 - 3(1+\rho)\Delta_{send\_min}/2 + \Theta$, com

tempo de espera para recuperação W igual a $(1+\rho)\pi/2 - (3-4\rho)\Delta_{send\_init}/2 + (1+4\rho)\Delta_{send\_max}/2 - 3\Delta_{send\_min}/2$.

PROVA. Considere-se um nodo que acabou de se tornar falho. O nodo falhou por *crash* após ter assumido o papel de testador ou de testado para cada um de seus enlaces adjacentes. Isto afeta tanto a latência de diagnóstico como o tempo de retenção de estado. De forma a computar o *tempo de retenção de estado para nodos no estado* não-respondendo, do inglês *node unresponsive state holding time*, SHT_{u_node} ambos os casos serão levados em conta.

Primeiro será considerado o caso no qual o nodo Y falha imediatamente após enviar uma requisição de teste, portanto tendo assumido o papel de testado para o intervalo de testes seguinte. Em outras palavras: Y falha no tempo $t + \Delta_{send\_init}$ imediatamente após enviar uma requisição de teste, a qual demora um intervalo de Δ_{send_min} para chegar ao nodo X. Se o testador X permanecer *sem-falha*, no intervalo de testes seguinte ele enviará uma requisição de teste para Y e haverá um *time-out* na resposta ao teste. X detectará o *time-out* após $t + \Delta_{send\_init} + \Delta_{send\_min} + (1+\rho)\pi + 2(1+2\rho)(\Delta_{send\_init} + \Delta_{send\_max})$.

Portanto o tempo mínimo que um nodo deve permanecer falho de forma a ter este estado diagnosticado deve ser maior ou igual à diferença entre o instante em que X detectou o *time-out* e o instante em que X recebeu a última requisição de teste de Y, o que resulta $(1+\rho)\pi + 2(1+2\rho)(\Delta_{send\_init} + \Delta_{send\_max})$. Chame-se a este resultado de γ_{tested}. Ele será útil mais tarde na prova do TEOREMA 8.

Supõe-se agora que o nodo Y se tornou falho no tempo $t + \Delta_{send\_init}$, após enviar uma requisição de teste para X, a qual é recebida no máximo no tempo $t + \Delta_{send\_init} + \Delta_{send\_max}$. Se Y recupera, ele enviará outra requisição de teste, a qual será recebida por X no mínimo no tempo $t + \Delta_{send\_init} + SHT + (1-\rho)W + \Delta_{send\_init} + \Delta_{send\_min}$.

A diferença entre os dois instantes de tempo nos quais X recebe requisições de teste de Y é $SHT + (1-\rho)W + \Delta_{send\_init} - \Delta_{send\_max} + \Delta_{send\_min}$. Esta diferença deve ser maior que γ_{tested} mais $D(\Delta_{send\_init} + \Delta_{send\_max}) + \Theta$ de forma a permitir que o evento de falha seja diagnosticado. Portanto $SHT_{u\_tested} > (1+\rho)\pi - (1-\rho)W + (D+1+4\rho)\Delta_{send\_init} + (D+3+4\rho)\Delta_{send\_max} - \Delta_{send\_min} + \Theta$.

De maneira análoga, considere-se um nodo Y que falha por *crash* após ter assumido o papel de testador para um enlace. Neste caso, se o vizinho envia uma requisição de teste no tempo t, o pior caso se dá quando o evento de falha ocorre no tempo $t + 2\Delta_{send\_init} + \Delta_{send\_min}$.

A diferença máxima entre os dois instantes de tempo em que respostas de testes consecutivas são recebidas pelo mesmo nodo é computada como segue. A primeira resposta a um teste enviada por Y é recebida por X no tempo $t + 2(\Delta_{send\_init} + \Delta_{send\_min})$ e a segunda resposta a um teste enviada por Y é recebida no tempo $t + 2(1 + \rho)\pi + 2(\Delta_{send\_init} + \Delta_{send\_max})$. Para que o evento de falha seja detectado, o *time-out* em X deve expirar no máximo no tempo $t + 2(1 + \rho)\pi + 2(1 + 2\rho)(\Delta_{send\_init} + \Delta_{send\_max})$. A diferença de tempo entre o instante em que o nodo X detecta um *time-out* e a primeira resposta a um teste é recebida é $2(1+\rho)\pi+4\rho\Delta_{send\_init}+2(1+2\rho)\Delta_{send\_max}-2\Delta_{send\_min}$. Chame-se a este resultado de γ_{tester}.

Suponha-se agora que o nodo Y se torna falho no tempo $t+2\Delta_{send\_init}+\Delta_{send\_max}$ após ter recebido uma requisição de teste e ter respondido. A resposta ao teste é recebida por X no máximo após Δ_{send_max}. Isto resulta $t+2(\Delta_{send\_init}+\Delta_{send\_max})$. A requisição de teste enviada por Y após recuperar é recebida por X no mínimo no tempo $t+2\Delta_{send\_init}+\Delta_{send\_max}+SHT+(1-\rho)W+\Delta_{send\_init}+\Delta_{send\_min}$. A diferença entre estes dois instantes de tempo deve ser maior que γ_{tester} mais $D(\Delta_{send\_init}+\Delta_{send\_max})+\Theta$ de forma a permitir que o evento seja diagnosticado. Isto resulta $SHT_{u\_tester} > 2(1 + \rho)\pi - (1 - \rho)W + (D - 1 + 4\rho)\Delta_{send\_init} + (D + 3 + 4\rho)\Delta_{send\_max} - 3\Delta_{send\_min} + \Theta$.

Como $SHT_{u\_tester} > SHT_{u\_tested}$, SHT_{u_tester} é empregado abaixo para calcular W e o tempo de retenção de estado final. Considere-se SHT_{w_node} como calculado no LEMA 12. Minimizando com respeito a W, tal como em [SUBBIAH e BLOUGH, 2004], pode ser concluído que ao passo em que W cresce, SHT_{w_node} monotonicamente também cresce, enquanto SHT_{u_tester} monotonicamente decresce. Para $W = 0$, $SHT_{w\_node} < SHT_{u\_tester}$. Portanto SHT_{w_node} e SHT_{u_tester} cruzam em algum ponto $W > 0$ e o tempo de retenção de estado é minimizado quando $SHT_{u\_tester} = SHT_{w\_node}$. Resolvendo para W obtemos o *tempo de espera para recuperação* mencionado no LEMA. Subtituindo na expressão

calculada acima para SHT_{u_tester} resulta no *tempo de retenção de estado para nodos no estado sem-falha*, do inglês *node unresponsive state holding time* SHT_{u_node} enunciado. □

LEMA 15. O tempo mínimo que um enlace deve permanecer no estado falho de forma a ser diagnosticado como *não-respondendo* é $2(1+\rho)\pi + (D+4\rho)\Delta_{send\_init} + (D+2+4\rho)\Delta_{send\_max} - 2\Delta_{send\_min} + \Theta$.

PROVA. Analogamente ao LEMA 13 acima, o *tempo de retenção de estado para enlaces no estado falho*, do inglês *link unresponsive state holding time*, SHT_{u_link}, deve ser maior que a latência máxima para diagnosticar um enlace que acabou de se tornar *não-respondendo*, como calculado no LEMA 10. □

4.2.3 Latência de Inicialização

A seguir a latência de inicialização para o algoritmo é calculada.

LEMA 16. A latência de inicialização de um nodo executando o algoritmo DNR é igual à sua máxima latência de diagnóstico.

ESQUEMA DA PROVA. Da definição de *latência delimitada de inicialização*, um nodo X que recupera no tempo t e permanece no estado *sem-falha* até $t+S$ tém, neste instante, estados L-válidos para cada nodo e enlace no sistema. Note que, por definição, $S \geq L$. Nós provamos neste LEMA que, de fato, $S = L$.

Um nodo arbitrário X recuperando no tempo t entra no *tempo de espera para recuperação* W e no máximo no tempo $t + (1+\rho)W$ envia requisições de teste aos seus vizinhos. O nodo X adquire informação de diagnóstico de seus vizinhos tão logo receba respostas aos testes ou detecte *time-outs* em enlaces adjacentes no estado *não-respondendo*. Considerando a possibilidade de testes simultâneos, uma requisição de teste pode levar até $(1+\rho)\pi$ a mais para ser enviada.

Quando o nodo X recupera, existem dois tipos de eventos dos quais ele tém que se tornar ciente: eventos que já foram totalmente disseminados até o tempo $t + (1+\rho)W$ e eventos que tenham ocorrido mas cuja informação esteja ainda sendo disseminados. X recebe informação sobre os eventos que já tenham sido disseminados através de seus vizinhos tão logo ele receba mensagens de *healing*. Considere agora os eventos que estejam

ainda sendo disseminados no tempo $t + (1 + \rho)W$, ou mais tarde. Para X ter recebido informações sobre os estados L-válidos de qualquer nodo ou enlace do sistema no tempo $t + S$, considere um evento arbitrário que tenha ocorrido em um enlace *não-adjacente* Y–Z no tempo t. Usando o fato de que L é da ordem de 2π enquanto W é da ordem de $\pi/2$, investigamos primeiramente se $S = L$; neste caso, o mais cedo que um evento pode ocorrer de forma a ser L-válido no tempo $t + S$ é no tempo t. Demonstramos que $S = L$. A prova deste LEMA encontra-se no Apêndice. □

4.2.4 As Três Propriedades de *Bounded Correctness*

TEOREMA 8. Com $W = (1 + \rho)\pi/2 - (3 - 4\rho)\Delta_{send\_init}/2 + (1 + 4\rho)\Delta_{send\_max}/2 - 3\Delta_{send\_min}/2$ e *tempos de retenção de estado* de $(1+\rho)W+(1+\rho)\pi+(D+2)(\Delta_{send\_init}+\Delta_{send\_max})+\Theta$ para nodos sem-falha e $2(1+\rho)\pi+(D+1)\Delta_{send\_init}+(D+2)\Delta_{send\_max}-\Delta_{send\_min}+\Theta$ para enlaces sem-falha, e *tempos de retenção de estado* de $(3+4\rho)\pi/2+(2D+1+\rho)\Delta_{send\_init}/2+(2D+5(1+\rho))\Delta_{send\_max}/2-3(1+\rho)\Delta_{send\_min}/2+\Theta$ para nodos falhos e $2(1 + \rho)\pi + (D + 4\rho)\Delta_{send\_init} + (D + 2 + 4\rho)\Delta_{send\_max} - 2\Delta_{send\_min} + \Theta$ para enlaces falhos, o algoritmo DNR atinge *bounded correctness* com latências de diagnóstico e de inicialização iguais a $max(2(1 + \rho)\pi + (D + 4\rho)\Delta_{send\_init} + (D + 2 + 4\rho)\Delta_{send\_max} - \Delta_{send\_min} + \Theta, 2(1 + \rho)\pi + (D + 1)\Delta_{send\_init} + (D + 2)\Delta_{send\_max} - \Delta_{send\_min} + \Theta)$.

PROVA.

PARTE 1: LATÊNCIA DELIMITADA DE DIAGNÓSTICO

Caso 1a: Evento de Falha. Inicialmente considere o seguinte evento: o enlace Y–Z se torna *não-respondendo* no tempo t enquanto o nodo X permanece continuamente *sem-falha* durante o intervalo $[t, t + L]$, onde L é a latência de diagnóstico. Para um enlace se tornar *não-respondendo*, tanto um nodo adjacente, ou o próprio enlace ou ambos, nodo e enlace se tornam falhos. Com os tempos de retenção de estado calculados, é demonstrado que um nodo X se torna ciente do estado de *não-respondendo* do enlace Y–Z no tempo $t + L$.

A maior latência possível de diagnóstico para um evento em que um enlace se torna *não-respondendo* ocorre quando um nodo falha como testador, como demonstrado

no LEMA 11. De fato, o SHT_{u_node} calculado no LEMA 14 leva este fato em conta. Com o W mencionado e usando o *tempo de retenção de estado para nodos falhos* calculado no LEMA 14, é demonstrado que o intervalo $SHT_{u\_node} + (1+\rho)W$ é suficiente para permitir a detecção do evento em que um enlace se torna *não-respondendo* antes que o nodo falho retorne ao estado *sem-falha* e envie requisições de teste.

Primeiramente considere que o nodo Y se torna falho no tempo $t + \Delta_{send\_init}$ após responder com sucesso a uma requisição de teste previamente enviada pelo nodo X. O nodo X recebe a resposta ao teste no máximo no tempo $t + \Delta_{send\_init} + \Delta_{send\_max}$. Após se tornar falho como testador do enlace, o nodo Y retorna ao estado *sem-falha* depois de $t + \Delta_{send\_init} + SHT_{u\_node}$. Uma requisição de teste é enviada não antes de $(1-\rho)W$ e chega ao nodo X após $\Delta_{send\_init} + \Delta_{send\_min}$. Subtraindo o tempo de chegada da resposta ao teste enviada por Y antes de se tornar falho do tempo de chegada desta última requisição de teste resulta $\gamma_{tester} + D(\Delta_{send\_init} + \Delta_{send\_max}) + \Theta$, que garante que o evento é diagnosticado.

Como γ_{tester} é computado considerando que o nodo Y se torna falho Δ_{send_min} antes que a primeira resposta ao teste chegue ao nodo X, a latência de detecção é igual a $\gamma_{tester} + \Delta_{send\_min}$. Adicionando o tempo de disseminação mais Θ resulta na *latência delimitada de diagnóstico* enunciada.

No caso de um nodo se tornar falho como nodo testado, o pior caso de latência ocorre se o nodo Y se torna falho no tempo $t + \Delta_{send\_init}$ após enviar uma requisição de teste. O nodo se torna o nodo a ser testado para o próximo intervalo de testes, e uma requisição de teste chega ao nodo X após Δ_{send_max}. Com um raciocínio semelhante ao caso acima, é demonstrado que a subtração dos tempos de chegada da requisição de teste enviada pelo nodo Y antes de se tornar falho da requisição de teste enviada pelo mesmo nodo após recuperar é maior que $\gamma_{tested} + D(\Delta_{send\_init} + \Delta_{send\_max}) + \Theta$. Portanto, o intervalo é suficiente para o nodo X detectar um *time-out* esperando por uma resposta a um teste vinda de Y antes de receber uma requisição de teste que indica que aquele nodo recuperou. Da mesma forma, é garantido que os outros nodos também recebem informação sobre o novo evento.

A latência de diagnóstico neste caso é igual a $\gamma_{tested} + \Delta_{send\_max} + D(\Delta_{send\_init} + \Delta_{send\_max}) + \Theta$, que é exatamente o resultado obtido no LEMA 10 para um nodo que se torna falho como nodo testado. Como esta latência é menor que a latência de diagnóstico mencionada neste TEOREMA, esta latência é limitada.

Finalmente, se o enlace *Y–Z* se torna *não-respondendo* porque o próprio enlace falhou, então SHT_{u_link} calculado no LEMA 15 garante que o evento seja diagnosticado com sucesso. É demonstrado no LEMA 10 que a latência de diagnóstico neste caso está dentro do limite de latência mencionado neste *teorema*.

Caso 1b: Evento de Healing. Considere um enlace arbitrário *Y–Z* que recupera no tempo t. Com um *tempo de espera para recuperação* W como especificado no TEOREMA e o *tempo de retenção de estado para nodos no estado sem-falha* fornecido, o *healing* de um enlace *Y–Z* devido à recuperação de um nodo Y é diagnosticado por qualquer nodo alcançável X dentro do limite de tempo $t + L$, como é demonstrado no LEMA 11.

Além disso, com o *tempo de retenção de estado para enlaces no estado sem-falha* como especificado no TEOREMA, um nodo arbitrário X diagnostica no tempo $t + L$ o *healing* de um enlace alcançável *Y–Z* ocorrido devido à recuperação do próprio enlace, como é também demonstrado no LEMA 11.

Uma vez que ambos os eventos de falha e de recuperação são diagnosticados no tempo mencionado no TEOREMA, a *Latência Delimitada de Diagnóstico* está provada para o algoritmo DNR.

PARTE 2: LATÊNCIA DELIMITADA DE INICIALIZAÇÃO

Considere um nodo X que entra no estado *sem-falha* no tempo t e um nodo Y e enlace *Y–Z* arbitrários para os quais o nodo X mantém um particular estado no tempo $t + S$, onde S é a latência de inicialização mencionada. É demonstrado abaixo que o estado mantido no tempo $t + S$ é L-válido.

No DNR, o estado mantido para um nodo pode ser tanto *sem-falha* como *inatingível*, enquanto que o estado mantido para um enlace pode ser tanto *sem-falha*, como *não-respondendo* ou *inatingível*. Para que um nodo mantenha o estado *não-respondendo* para um enlace alcançável, pelo menos o próprio enlace ou um de seus nodos adjacentes

devem estar falhos. O estado *inatingível* para um nodo ou enlace é deduzido do estado *não-respondendo* mantido para um ou mais enlaces. Portanto um enlace pode ser considerado *inatingível* quando ele é não adjacente a nenhum nodo alcançável. Um nodo é considerado *inatingível* quando todos os seus enlaces adjacentes são *não-respondendo* ou *inatingíveis*. Por outro lado, para que um nodo mantenha o estado *sem-falha* para um nodo ou enlace alcançável, o próprio enlace e ambos os nodos adjacentes devem estar sem-falha.

Prova-se que não é possível para um nodo X manter o estado *sem-falha* para outro nodo Y e enlace Y–Z no tempo $t+S$ se aquele nodo e enlace permanecem *não-respondendo* ou *inatingíveis* durante o intervalo $[t, t+S]$ inteiro. Além do mais, não é possível para um nodo X manter os estados *não-respondendo* ou *inatingível* para um nodo Y ou enlace Y–Z no tempo $t+S$ se aquele nodo e enlace permanecem *sem-falha* durante o intervalo $[t, t+S]$ inteiro.

Possibilidade 2a: X mantém o estado sem-falha *para o enlace Y–Z mas o enlace Y–Z permanece* não-respondendo *durante todo o intervalo.*

Inicialmente suponha-se que $Z = X$, de forma que o enlace Y–Z seja adjacente ao nodo X. Se o nodo X recupera no tempo t, no tempo $t+S$, X mantém o estado *sem-falha* para o nodo Y e enlace Y–Z somente se ele recebe uma requisição de teste e a subsequente mensagem com informações da tabela de enlaces do nodo Y ou uma resposta de teste do mesmo nodo durante $[t, t+S]$. Uma vez que o tempo $t+(1+\rho)W+2(\Delta_{send\_init}+\Delta_{send\_max})$ é anterior a $t+S$, a única forma de X receber uma resposta de teste de Y é se o nodo Y responde à requisição de teste que X enviou ao recuperar. Se isto ocorre, o enlace Y–Z certamente não estará *não-respondendo* como assumido, então esta possibilidade não pode ocorrer.

A única possibilidade para o nodo X receber uma requisição de teste do nodo Y durante $[t, t+S]$ se o nodo Y está falho durante todo o intervalo é se o nodo Y se torna falho no tempo t imediatamente após ter enviado uma requisição de teste. Se o enlace está sem-falha, a requisição de teste chega a X no máximo no tempo $t+\Delta_{send\_max}$, enquanto o nodo X está ainda no *tempo de espera para recuperação*. No tempo $t+(1+\rho)W$,

quando o nodo X finalmente testa, o nodo Y está em $SHT_{u\_node} + (1-\rho)W$. Portanto o nodo X detecta um *time-out* após $2(1+2\rho)(\Delta_{send\_init} + \Delta_{send\_max})$, dentro da latência de inicialização anunciada. Por outro lado, se o próprio enlace Y–Z se torna falho no tempo t, então no tempo em que o nodo X testa, o enlace está em SHT_{u_link} e o nodo X também detecta um *time-out*.

Até este ponto foram considerados enlaces adjacentes. Para enlaces não adjacentes, $Z \neq X$. O nodo X mantém o estado *sem-falha* para o nodo Y e enlace Y–Z no tempo $t+S$ somente se recebe mensagens de *healing* dos vizinhos ou mensagens de disseminação com esta informação. Além disso, em ambos os casos não pode haver nenhuma mensagem subsequente de disseminação durante $[t, t+S]$ com informação de que o enlace Y–Z está *não-respondendo*. Demonstramos que estes casos não ocorrem se Y–Z permanece *não-respondendo* durante todo o intervalo $[t, t+S]$.

Se o nodo X entra no estado *sem-falha* no tempo t, ele recebe mensagens de *healing* de seus vizinhos no máximo no tempo $t + (1+\rho)W + (1+\rho)\pi + 2(\Delta_{send\_init} + \Delta_{send\_max})$. Como este intervalo de tempo é anterior a $t+S$, se os vizinhos do nodo X mantém o estado *sem-falha* para o enlace Y–Z naquele tempo, a informação sobre seu estado *não-respondendo* está sendo disseminada. Como foi demonstrado no LEMA 16, no máximo no tempo $t+S$ esta disseminação estará completa.

Agora demonstra-se que o nodo X não pode receber uma mensagem de disseminação atribuindo o estado *sem-falha* para o enlace Y–Z. Antes de tornar-se *não-respondendo*, o enlace Y–Z ou estava previamente no estado *sem-falha* ou tinha estado *não-respondendo* desde que os outros nodos inicializaram pela primeira vez. No primeiro caso, o enlace Y–Z permaneceu *sem-falha* pelo menos por SHT_{w_node}, que é tempo suficiente para que esta informação seja completamente disseminada.

Portanto se o enlace Y–Z está *não-respondendo* durante o intervalo $[t, t+S]$ inteiro, então não é possível para o nodo X manter o estado *sem-falha* para o nodo Y e enlace Y–Z no tempo $t+S$.

Possibilidade 2b: X mantém o estado não-respondendo *para o enlace* Y–Z *mas o enlace* Y–Z *permanece* sem-falha *durante todo o intervalo.*

Considere-se um nodo arbitrário X que entra no estado *sem-falha* no tempo t. Primeiramente, suponha que $Z = X$, de forma que o enlace Y–Z seja adjacente ao nodo X. O nodo X mantém o estado *não-respondendo* para aquele enlace somente se ele não recebe de Y nem uma requisição de teste nem uma resposta a um teste durante o intervalo $[t, t + S]$. O único modo para que o nodo X não receba uma resposta a um teste ou uma requisição de teste do nodo Y é se o enlace Y–Z está *não-respondendo* durante todo o intervalo. Se o enlace Y–Z está *sem-falha* durante todo o intervalo como assumido, então o nodo Y vai responder à requisição de teste de X no máximo no tempo $\Delta_{send\_init} + \Delta_{send\_max}$ após $t + (1 + \rho)W$. Esta resposta chegará ao nodo X no máximo após o tempo $\Delta_{send\_init} + \Delta_{send\_max}$, e portanto dentro do limite da latência de inicialização enunciado.

Se, no pior caso, o nodo Y recupera no tempo t então ambos os nodos X e Y estão no *tempo de espera para recuperação* simultaneamente. Portanto, quando eles mandam requisições de teste no tempo $t + (1 + \rho)W$, testes simultâneos podem ocorrer e o nodo X se torna ou o nodo testador ou o nodo testado. Se X se torna o testador, ele leva até $t + (1 + \rho)W + 2(\Delta_{send\_init} + \Delta_{send\_max})$ para reconhecer que o enlace está *sem-falha*, como no caso acima. Se o nodo X se torna o nodo testado então no máximo no tempo $\Delta_{send\_init} + \Delta_{send\_max}$ mais $t + (1 + \rho)W + 2(\Delta_{send\_init} + \Delta_{send\_max})$ ele recebe uma mensagem de *healing* do nodo Y com o *timestamp* do enlace incrementado, confirmando a ocorrência do evento de *healing*.

Para enlaces não adjacentes, o nodo X mantém o estado *não-respondendo* para o enlace Y–Z somente se ele recebe ou uma mensagem de *healing* ou uma mensagem de disseminação com informação de que o enlace Y–Z está *não-respondendo*. Em ambos os casos, nenhuma mensagem de disseminação com a informação oposta pode ser recebida durante $[t, t + S]$. Demonstramos que estes casos não podem ocorrer se o enlace Y–Z permanece *sem-falha* durante o intervalo $[t, t + S]$ inteiro.

Se o nodo X entra no estado *sem-falha* no tempo t, ele recebe mensagens de *healing* de seus vizinhos no máximo no tempo $t + (1 + \rho)W + (1 + \rho)\pi + 2(\Delta_{send\_init} + \Delta_{send\_max})$. Como aquele intervalo de tempo é anterior a $t + S$, se os vizinhos do nodo X mantém o

estado *não-respondendo* para o enlace *Y–Z* durante aquele tempo, a informação sobre o *healing* daquele enlace está sendo disseminada. No máximo no tempo $t + S$, o evento de *healing* estará completamente diagnosticado, como demonstrado no LEMA 16.

Demonstra-se agora que o nodo X não pode receber uma mensagem de disseminação atribuindo o estado *não-respondendo* ao enlace *Y–Z*, porque o enlace recupera necessariamente após ter permancecido *não-respondendo* por SHT_{u_node} ou tém estado *não-respondendo* desde a primeira inicialização dos demais nodos. No primeiro caso, o enlace *Y–Z* permaneceu *não-respondendo* por no mínimo $SHT_{u\_node} + (1 - \rho)W$, que é tempo suficiente para que esta informação seja completamente disseminada.

Portanto se o enlace *Y–Z* esteve *sem-falha* durante $[t, t + S]$, no tempo $t + S$ não é possível ao nodo X manter o estado *não-respondendo* para aquele enlace.

PARTE 3: ACCURACY

Um nodo X no estado *sem-falha* executando o algoritmo DNR muda o estado de um enlace *Y–Z* de *sem-falha* para *não-respondendo* somente quando X tem o estado *sem-falha* atribuído para aquele enlace e uma de duas situações ocorre: ou detecta o enlace como *não-respondendo* após um *time-out* esperando por uma resposta a um teste ou recebe uma mensagem de disseminação que causa a atribuição do estado *não-respondendo* para aquele enlace. A latência máxima para diagnosticar um enlace como *não-respondendo* é menor que $SHT_{u\_node} + (1 - \rho)W$ mais o tempo mínimo que leva para uma requisição de teste chegar a um vizinho, isto é, $\Delta_{send\_init} + \Delta_{send\_min}$. O estado atribuído por X no tempo t para o enlace *Y–Z* é portanto L-válido, isto é, o enlace se tornou *não-respondendo* no tempo $t - L$ ou mais tarde e permaneceu naquele estado por tempo suficiente para ser diagnosticado.

De maneira similar, um nodo X no estado *sem-falha* muda o estado de um enlace *Y–Z* de *não-respondendo* para *sem-falha* somente quando X tem o estado *não-respondendo* atribuído para aquele enlace e detecta que o enlace está *sem-falha* pelo recebimento de uma resposta a um teste ou de uma mensagem de disseminação, a qual causa a atribuição do estado *sem-falha* para aquele enlace. Como a latência máxima para diagnosticar um evento de *healing* após a recuperação de um nodo é menor que SHT_{w_node}, e a mesma

latência após a recuperação de um enlace é menor que SHT_{w_link}, o estado assinalado é L-válido.

Portanto, assumindo que o estado inicial assinalado para qualquer enlace é válido, como demonstrado na Parte 2, qualquer transição de estado armazenada por um nodo X para qualquer nodo Y e enlace Y–Z é também válido e a propriedade de *acuidade* é garantida. □

CAPÍTULO 5

RESULTADOS EXPERIMENTAIS

Neste Capítulo são apresentados resultados de simulação do algoritmo DNR. Inicialmente, o ambiente de simulação utilizado é detalhado. Nos primeiros resultados experimentais particionamentos na rede não ocorrem. Para estes experimentos a ocorrência de eventos em nodos e a ocorrência de eventos em enlaces é simulada de maneira indepentente, e os resultados são apresentados em duas Subseções. Em ambos os casos, o desempenho do algoritmo é avaliado tanto em eventos de falha como em eventos de recuperação e três diferentes tipos de topologias são utilizadas. Em seguida, o desempenho do algoritmo é avaliado em situações na qual a rede sofre particionamento. Para este tipo de experimento, são utilizadas topologias do tipo *Power-Law*. Um terceiro conjunto de experimentos é apresentado no qual um dos parâmetros do algoritmo, π, ou o intervalo de testes é avaliado. Finalmente, é apresentada uma Seção com resultados relativos ao número de mensagens de disseminação utilizado pelo algoritmo.

5.1 AMBIENTE DE SIMULAÇÃO

O programa de simulação foi construído usando a biblioteca de simulação de eventos discretos SMPL [MacDougall, 1987]. Os experimentos foram realizados no *cluster* de alto desempenho da Universidade Federal do Paraná, composto por cerca de 60 máquinas, as principais com 16 processadores e 16 GB de memória cada, bem como máquinas com 8 processadores e 8 GB de memória cada. Os processadores incluem *Opteron* 1.8 GHz com 1MB de *cache* L2, Intel *quad-core* 2.4 GHz e Athlon, conectados por rede *Myrinet* e *Gigabit-Ethernet*. O ambiente nas máquinas é exclusivamente de 64 bits, baseado no sistema operacional Linux. Há um servidor central de disco que permite *boot* remoto.

Os experimentos foram conduzidos de forma a medir a latências de diagnóstico de eventos de falha e de recuperação tanto em nodos como em enlaces. De forma a

obter um intervalo de confiança igual a 95% para as médias das latências, cinco rodadas independentes foram inicialmente executadas para um dado tipo e tamanho de topologia; 5000 eventos de falha e de recuperação (2500 de cada tipo) foram escalonados em cada rodada. A semi-amplitude do intervalo de confiança foi avaliada até ser menor ou igual a 10% da média obtida para todas as rodadas. O dinamismo da ocorrência de eventos foi dado por um processo de Poisson. Tão logo um evento ocorre em um nodo ou em um enlace, o evento seguinte no mesmo nodo ou enlace é escalonado para ocorrer após o tempo de retenção de estado correspondente mais um período de tempo exponencialmente distribuído. Os eventos são concorrentes, escalonados para ocorrer em qualquer nodo ou enlace. Duas médias para o processo de Poisson foram utilizadas: 1 segundo e 200 segundos, portanto foram simulados sistemas altamente dinâmicos. A menor média da distribuição exponencial corresponde a um ambiente mais dinâmico, onde nodos mudam de estado com mais frequência, pouco tempo depois de permanecer em um estado pelo tempo de retenção de estado correspondente. A média de 200 segundos para a distribuição exponencial simula o ambiente oposto, no qual os nodos permanecem em cada estado por mais tempo.

Um conjunto de experimentos foi conduzido de forma a comparar o desempenho dos algoritmos DNR e *ForwardHeartbeat*. Nestes experimentos, os parâmetros de simulação e o dinamismo da ocorrência de eventos são similares àqueles utilizados em [Subbiah e Blough, 2004].

Os parâmetros de simulação estão listados na Tabela 5.1. Os parâmetors Δ_{send_init}, Δ_{send_min} e Δ_{send_max} correspondem, respectivamente, ao tempo para um nodo iniciar uma comunicação e aos tempos mínimo e máximo de atraso num canal de comunicação. Seus valores foram mantidos fixos em todos os experimentos.

5.2 TOPOLOGIAS UTILIZADAS NOS EXPERIMENTOS

Os experimentos foram executados em quatro tipos de topologias. Grafos aleatórios com diferentes conectividades de vértices; grafos aleatórios com distribuição *Power-Law* e topologias regulares dos tipos *mesh* e hipercubo. As topologias são descritas a seguir. O

Tabela 5.1: Parâmetros de simulação.

Parâmetros de Simulação	
Simulador	SMPL
Δ_{send_init}	0.002 seconds
Δ_{send_min}	0.008 seconds
Δ_{send_max}	0.08 seconds
π	30 seconds
ρ	0.0001 seconds
Recovery wait time W	15.026516 seconds
Médias para o processo de Poisson	1 segundo, 200 segundos
# de rodadas iniciais	5
# de eventos por rodada	5000 (2500 de falha e 2500 de recuperação)
Intervalo de Confiança	95%

conjunto de topologias é mostrado na Tabela 5.2.

Tabela 5.2: Topologias utilizadas nos experimentos.

Topologias	
Grafos aleatórios com conectividade de vértices $k = 3$	32, 64, 128, 256 nodos
Grafos aleatórios com conectividade de vértices $k = \log n$	32, 64, 128, 256 nodos
Grafos aleatórios com distribuição *Power-Law*	32, 64, 128, 256 nodos
Topologias *Mesh*	64, 256 nodos
Hipercubos	32, 64, 128, 256 nodos

Primeiramente, grafos com conectividade de vértices igual a k gerados aleatoriamente foram obtidos como em [Subbiah e Blough, 2004]. A conectividade de vértices (arestas) de um grafo é o menor número de vértices (arestas) cuja remoção pode desconectar o grafo. Uma topologia inicial de n nodos com grau k foi obtida aleatoriamente. A conectividade de vértices da topologia foi então repetidamente avaliada. A cada vez em que a conectividade desejada não era atingida, n enlaces adicionais eram aleatoriamente introduzidos, até que uma topologia com conectividade de vértices igual a k era obtida. Topologias foram geradas para n igual a 32, 64, 128 e 256 nodos com $k = 3$ e $k = \log n$; todos os logaritmos deste trabalho são em base 2. Cinco diferentes topologias foram geradas para cada valor de n e k.

Topologias com distribuição *Power-Law* refletem a topologia da própria Internet [Bu e Towsley, 2002], onde poucos nodos possuem muitos enlaces e a imensa maioria dos nodos possuem poucos enlaces. Grafos aleatórios com a distribuição *Power-Law* foram obtidas utilizando o gerador de [Bu e Towsley, 2002]. Naquele gerador, o número total n de nodos e o número inicial de nodos no *backbone* são dados. A cada iteração, novos nodos são criados com dada probabilidade p. Com probabilidade $1 - p$, novos enlaces são

criados. Neste caso, os nodos que serão conectados pelo novo enlace são escolhidos de acordo com um parâmetro, $\beta \in (-\infty, 1)$, o qual define o grau de preferência por nodos com muitos vizinhos. Quanto menor o valor de β, menor a preferência dada por um enlace a nodos com grau alto. Com $p = 0.6$ e $\beta = 0.2$, cinco diferentes topologias foram geradas para cada número n de 32, 64, 128 e 256 nodos. Estas topologias foram todas avaliadas como tendo conectividades de vértices e arestas iguais a 1, sendo, portanto, adequadas a experimentos com particionamento da rede.

Além destas, as topologias regulares *mesh 8x8* e *16x16* [Culler e Singh, 1999] e hipercubos com 32, 64, 128 e 256 nodos foram também empregadas nos experimentos de simulação.

Para cada uma das topologias utilizadas nos experimentos, suas conectividades de vértices e de arestas foram avaliadas utilizando o algoritmo clássico de Ford-Fulkerson [Cormen, Leiserson, Rivest *et. al.*, 2001]. A conectividade da rede é utilizada para construir dois cenários de simulação: o cenário no qual particionamentos da rede não ocorrem e o cenário no qual a rede pode se tornar particionada. Eventos ocorrendo exclusivamente em nodos e exclusivamente em enlaces foram simulados em ambos os cenários.

Ao simular eventos em nodos no cenário em que particionamentos não ocorrem, o número máximo de $k - 1$ nodos falhos simultaneamente foi permitido, tendo a rede conectividade de vértices igual a k. De maneira similar para eventos em enlaces e redes com conectividade de arestas igual a k. Neste cenário, se o número de nodos ou enlaces falhos é igual a $k-1$ quando um evento de falha em nodo ou enlace vai ocorrer, um período de tempo exponencialmente distribuído é utilizado para reescalonar aquele evento para um instante de tempo posterior.

Por outro lado, ao simular o cenário no qual a rede pode sofrer particionamentos e reconexões, eventos de falha são permitidos concorrentemente em número maior que o da conectividade da rede.

Como cinco diferentes topologias aleatórias de cada tipo foram geradas para cada número n de nodos, uma rodada foi simulada em cada topologia, e as médias das cinco

rodadas foram testadas para a precisão do intervalo de confiança desejada. Com topologias regulares, como apenas uma topologia de cada tipo existe para cada número n de nodos, cinco rodadas independentes foram obtidas variando a semente para geração de números aleatórios em cada rodada.

Os experimentos são apresentados a seguir em quatro Seções; na primeira Seção são apresentados experimentos sem particionamento da rede; na segunda Seção são apresentados resultados de simulação em que particionamentos podem ocorrer. Em ambas, o intervalo de testes π é mantido sempre como 30 segundos. A Seção seguinte avalia o desempenho do algoritmo em situações em que o intervalo de testes é da ordem do atraso no canal de comunicação. A Seção final tece considerações sobre o número de mensagens de disseminação utilizado pelo algoritmo.

5.3 EXPERIMENTOS SEM PARTICIONAMENTO DA REDE

Para avaliar o desempenho do algoritmo, as ocorrências de eventos em nodos e as ocorrências de eventos em enlaces foram simuladas de maneira independente. Em ambos os casos, três tipos de topologias foram avaliadas: grafos aleatórios com diferentes conectividades e topologias regulares dos tipos *mesh* e hipercubo. Os resultados são apresentados a seguir, em duas Subseções.

5.3.1 Experimentos com Eventos em Nodos

Nesta Subseção são apresentados os resultados de simulação de eventos de falha e de recuperação exclusivamente em nodos. O primeiro conjunto de experimentos utiliza grafos aleatórios com conectividades de vértices iguais a 3 e $\log n$. Este conjunto de experimentos permitiu comparar o desempenho dos algoritmos DNR e *ForwardHeartbeat*.

Em seguida, são apresentados os resultados de simulação de eventos em nodos nas topologias regulares dos tipos *mesh* e hipercubo.

5.3.1.1 Experimentos com Topologias Aleatórias

O primeiro conjunto de experimentos de simulação é análogo aos experimentos executados em [Subbiah e Blough, 2004], e permitiu a comparação com o algoritmo *ForwardHeartbeat*. Primeiramente, eventos de falha e de recuperação ocorrendo exclusivamente em nodos foram simulados com ambas as médias da distribuição exponencial de 1 segundo e de 200 segundos, em topologias com conectividade de vértices igual a 3. Os resultados das latências de diagnóstico são mostrados nas Figuras 5.1 e 5.2. Após isso, eventos de falha e de recuperação ocorrendo em nodos foram simulados em experimentos com média de 200 segundos para o processo de Poisson em ambas as redes com conectividade de vértices iguais a 3 e $\log n$. Os resultados são mostrados nas Figuras 5.3 e 5.4.

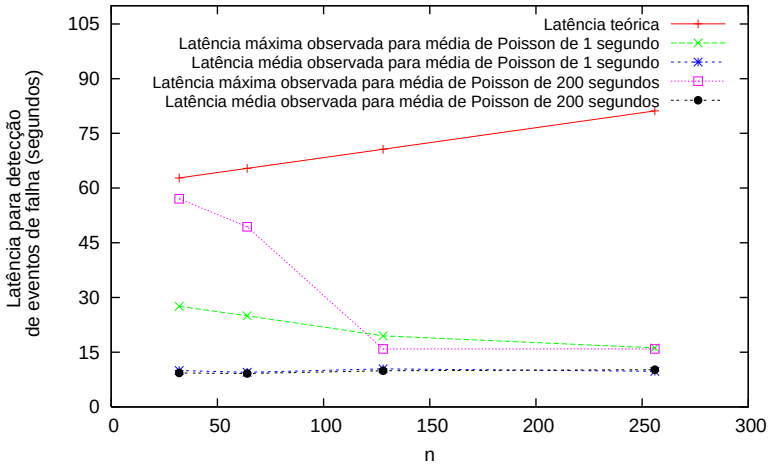

Figura 5.1: Eventos de falha em nodos - grafos aleatórios - conectividade de vértices igual a 3.

Como pode ser observado, as latências médias não são influenciadas nem pela conectividade da rede nem pelo dinamismo da ocorrência de eventos dado pelo processo de Poisson. Latências médias para eventos de falha são da ordem de um terço do intervalo de testes com conectividades da rede e médias da distribuição exponencial variadas. Comparando-se com os resultados de [Subbiah e Blough, 2004], as latências médias do *ForwardHeartbeat* são da ordem de dois terços do intervalo entre *heartbeats* nas mesmas condições. As latências máximas do DNR, por outro lado, mostram um acréscimo em redes pequenas. Isto é devido ao fato de que, com poucos nodos, é maior a probabilidade de que eventos de falha em nodos contíguos ocorram. Assim, é possível que o próximo

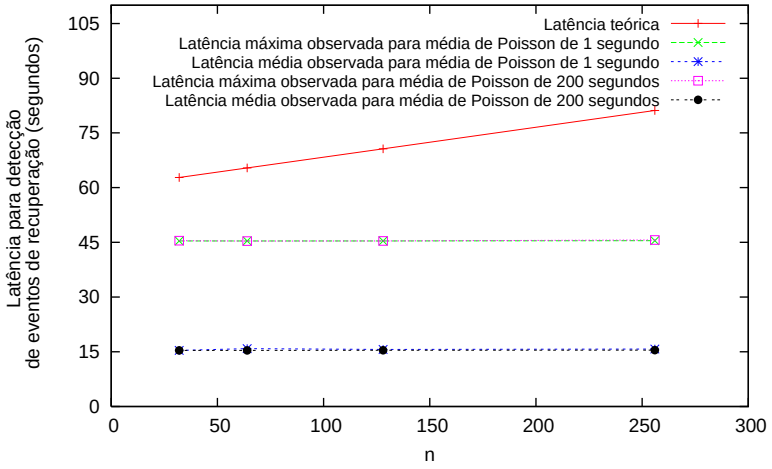

Figura 5.2: Eventos de recuperação em nodos - grafos aleatórios - conectividade de vértices igual a 3.

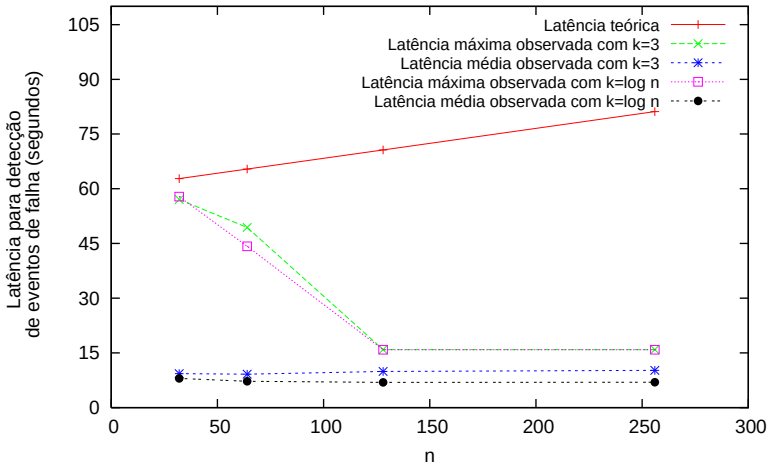

Figura 5.3: Eventos de falha em nodos - grafos aleatórios - média de Poisson de 200 segundos.

testador em um intervalo de testes esteja falho quando seu vizinho também falha. Com uma média de 200 segundos para o processo de Poisson, aquele testador permanece falho por mais tempo. Desta forma, pode levar até o máximo de dois intervalos de teste para que a falha do nodo vizinho, descrita acima, seja detectada por outro nodo.

Em experimentos de recuperação, a latência máxima ocorre em casos de testes simultâneos. Como estes casos são raros, eles não influenciam as latências médias, que são mantidas na ordem da metade do intervalo de testes. Além disto, em oposição aos experimentos com *ForwardHeartbeat*, a latência é calculada levando em conta o *recovery wait time*. De fato, tão logo expira o tempo de $W = 15.026516$ segundos, cada recuperação

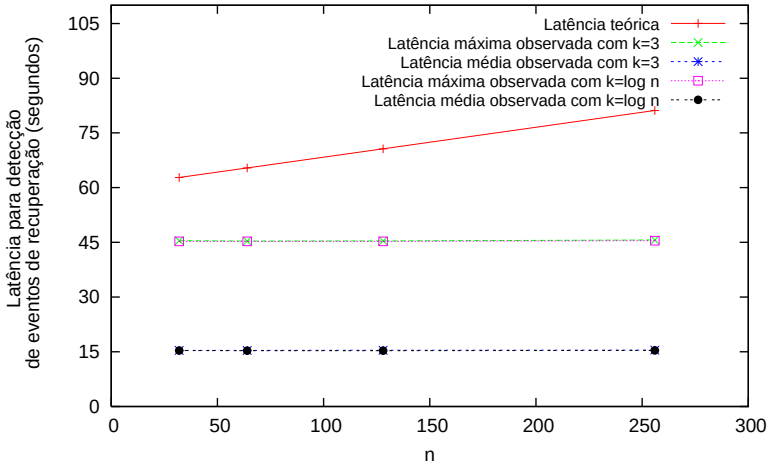

Figura 5.4: Eventos de recuperação em nodos - grafos aleatórios - média de Poisson de 200 segundos.

de um nodo é detectada. Se o tempo de espera para recuperação não fosse levado em conta, as latências de recuperação do DNR também seriam da ordem de décimos de segundos, isto é, da ordem apenas do tempo de disseminação do evento. As latências médias também não crescem com o acréscimo no número de nodos na rede porque o diâmetro da rede mantém-se pequeno em comparação com n. O diâmetro máximo observado é de 25 em uma rede de 256 nodos e é da ordem de 7 em redes com 32 nodos.

Em ambos os tipos de experimentos, a latência teórica não muda com diferentes parâmetros de simulação ou tipos de eventos e é uma função do número n de nodos na rede.

5.3.1.2 Experimentos com Topologias Regulares

Um segundo conjunto de experimentos com eventos em nodos foi executado em topologias regulares. As topologias foram redes *mesh* e hipercubos, e novamente as médias para o processo de Poisson foram de 1 segundo e 200 segundos. A conectividade de vértices de redes *mesh* é 4 e a conectividade de vértices de hipercubos é $\log n$. Um número máximo de nodos igual à conectividade menos 1 esteve falho simultaneamente em dado instante de tempo.

As latências médias de eventos de falha são novamente da ordem de um terço do intervalo de testes, como pode ser visto nas Figuras 5.5 e 5.6. Novamente as latências

máximas mostram um acréscimo em redes pequenas para experimentos com média da distribuição exponencial de 200 segundos. Com média da distribuição exponencial igual a 1 segundo, a latência máxima apresenta um pequeno acréscimo com o acréscimo do tamanho da rede.

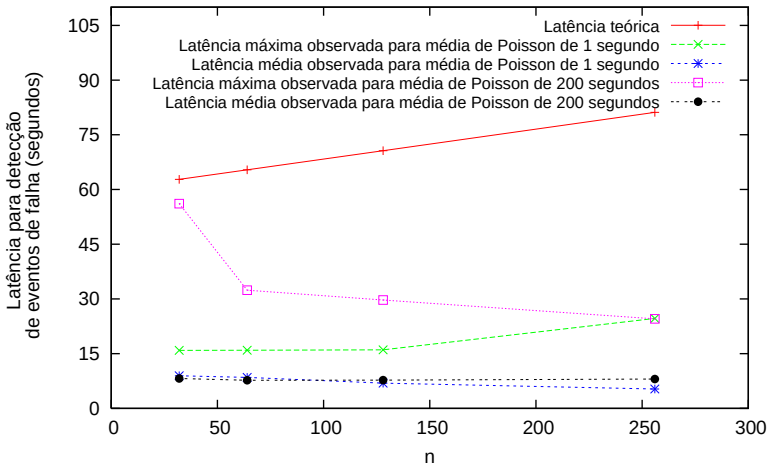

Figura 5.5: Eventos de falha em nodos - hipercubos.

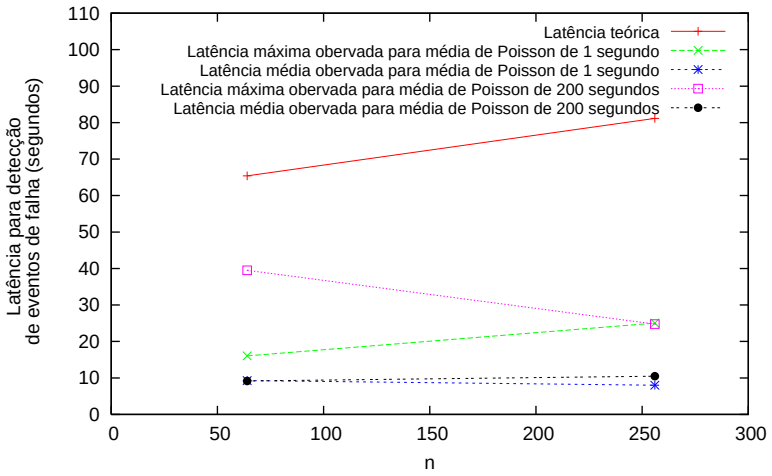

Figura 5.6: Eventos de falha em nodos - redes *mesh*.

Os gráficos dos experimentos de *healing* exibem exatamente as mesmas curvas que aqueles do primeiro conjunto de experimentos acima e portanto não serão mostrados aqui. De fato, como um evento de recuperação de um nodo é detectado tão logo aquele nodo conclua o tempo de espera para recuperação, nenhum mudança na topologia influencia estes resultados, desde que o nodo em recuperação tenha ao menos um vizinho para detectar o evento.

5.3.2 Experimentos com Eventos em Enlaces

Um outro conjunto de experimentos de simulação foi executado nas mesmas topologias acima. Para este tipo de experimentos, eventos foram escalonados para ocorrer exclusivamente em enlaces. Os resultados espelham o que é teoricamente esperado: sem eventos ocorrendo em nodos, um enlace falho ou recentemente recuperado tem sempre um testador, e qualquer detecção ocorre em no máximo um intervalo de testes, independente do tipo de evento. As médias, tanto para eventos de falha como para eventos de recuperação, são iguais à metade do intervalo de testes. Esta regularidade é mantida tanto para redes aleatórias com ambas as conectividades de 3 e $\log n$ como para as redes *mesh* e os hipercubos. Os gráficos para topologias aleatórias são mostrados nas Figuras 5.7 e 5.8.

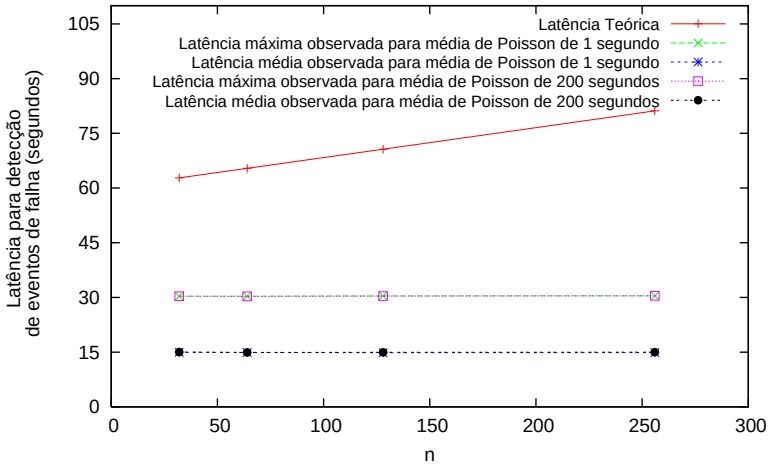

Figura 5.7: Eventos de falha em enlaces - grafos aleatórios - conectividade de vértices igual a $\log n$.

5.4 EXPERIMENTOS COM PARTICIONAMENTO DA REDE

Experimentos com particionamento da rede foram executados em grafos aleatórios gerados com a distribuição *Power-Law*. Um componente conexo de uma rede pode sofrer particionamento quando um enlace ponte sofre um evento de falha ou quando um nodo falha. No primeiro caso, a rede fica particionada em dois componentes conexos, contendo cada um deles um dos nodos conectados pelo enlace recentemente falho. Um dos nodos, o próximo testador, detectará o evento em no máximo um intervalo de testes, enquanto

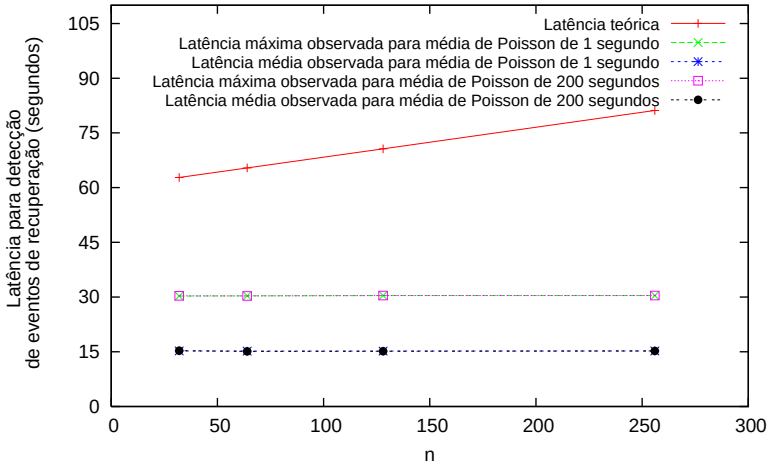

Figura 5.8: Eventos de recuperação em enlaces - grafos aleatórios - conectividade de vértices igual a $\log n$.

o nodo que seria testado caso o enlace se mantivesse sem-falha detectará o evento no intervalo de testes seguinte. Por outro lado, a falha de um nodo, sobretudo daqueles com grande número de enlaces, pode criar vários componentes conexos. Em cada componente conexo, os nodos vizinhos ao nodo falho também detectarão eventos de falha em seus enlaces em um ou dois intervalos de teste, conforme a configuração da rede no momento da falha. Como a falha de um nodo pode ser simulada pela falha em todos os seus enlaces, somente eventos em enlaces foram escalonados neste tipo de experimento. O número de eventos escalonado foi sempre maior que o da conectividade da rede, porém não tão grande que a rede se reduzisse a uma rede trivial.

Foram medidas nestes experimentos apenas as latências de diagnóstico de eventos de falha. Latências de eventos de recuperação não foram medidas, por estarem suficientemente representadas nos experimentos anteriores. Para cada evento, foram medidas a latência de diagnóstico ocorrida no primeiro intervalo de testes, em um dos novos componentes conexos, e a latência de diagnóstico ocorrida para o mesmo evento no segundo intervalo de testes, no outro componente conexo. Ambas as latências foram computadas separadamente para cada evento ocorrido, e as médias das latências do diagnóstico de eventos ocorridos em cada intervalo de testes são mostradas na Figura 5.9. A média utilizada no processo de Poisson foi de 200 segundos.

Como num grafo aleatório gerado com a distribuição *Power-Law* um grande número

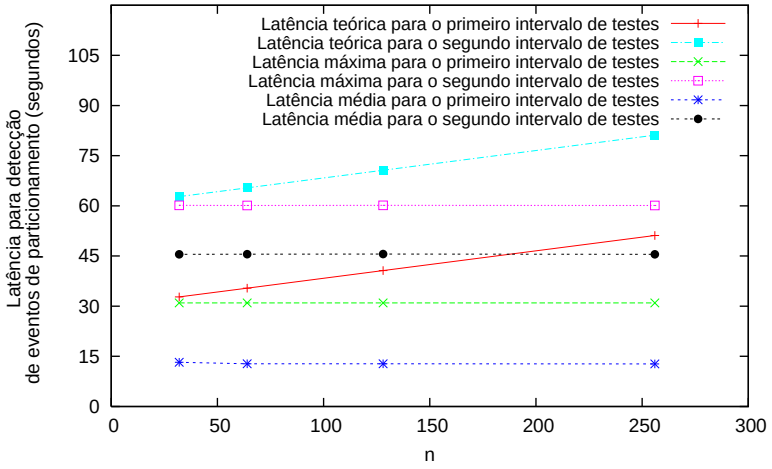

Figura 5.9: Eventos de falha em enlaces que particionam a rede - grafos *Power-Law*.

de enlaces ponte são enlaces de nodos de borda, em muitos casos a disseminação de mensagens de diagnóstico fica confinada a componentes conexos com apenas um nodo. Isto é compensado pelas disseminações que ocorrem nos componentes conexos formados pelo restante da rede e, na média, as latências exibem o mesmo padrão de comportamento encontrado nos experimentos com falhas em enlace. Desta forma, tanto as latências médias de diagnóstico ocorrido no primeiro como no segundo intervalo de testes se mantém na metade do intervalo de testes correspondente.

5.5 EXPERIMENTOS VARIANDO O INTERVALO DE TESTES

Nos experimentos mostrados até o momento, o intervalo de testes foi mantido igual a 30 segundos. Com os valores utilizados para atrasos nos canais de comunicação (Δ_{send_min} e Δ_{send_max}, conforme consta na Tabela 5.1), a contribuição da fase de disseminação na latência de diagnóstico é irrisória.

Assim, foi executado um conjunto de experimentos com a finalidade de mostrar a contribuição da fase de disseminação em experimentos com intervalos de testes menores.

Com efeito, mantendo os valores de Δ_{send_min} e Δ_{send_max} como acima, e atribuindo o valor de $0,5$ segundos para o intervalo de testes π, obteve-se os resultados que serão mostrados nesta Seção.

Os experimentos são de falhas em enlaces sem particionamento da rede e com média

de Poisson igual a 1 segundo. A Figura 5.10 mostra eventos simulados em hipercubos. Como o diâmetro de um hipercubo é igual a $\log n$, onde n é o número de nodos no grafo, a curva mostra uma linha ascendente discreta. Na Figura 5.11, entretanto, onde são mostrados os resultados para redes *mesh*, cujos diâmetros são maiores do que os diâmetros dos hipercubos, a curva ascendente é bem mais visível.

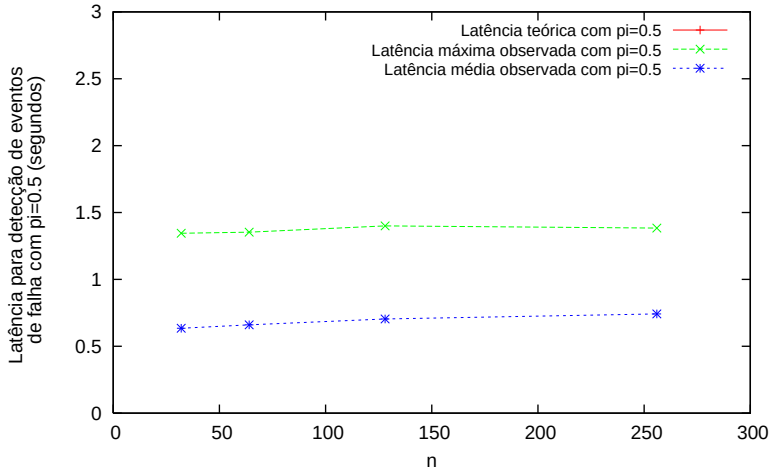

Figura 5.10: Eventos de falha em enlaces - hipercubos - $\pi = 0,5$ segundos.

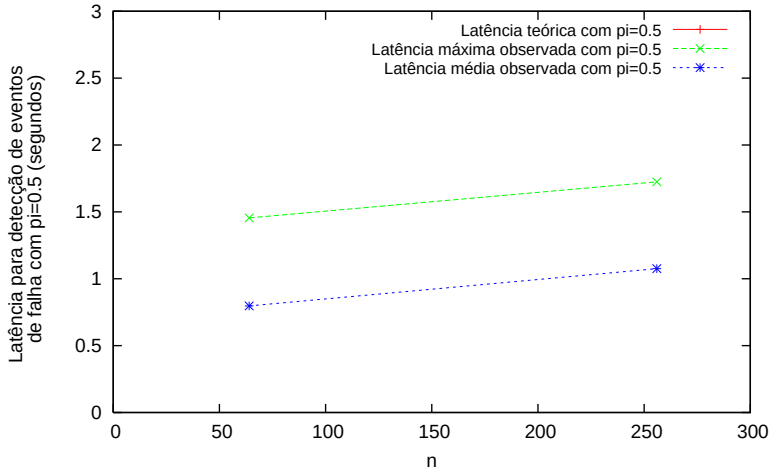

Figura 5.11: Eventos de falha em enlaces - redes *mesh* - $\pi = 0,5$ segundos.

As Figuras 5.12 e 5.13 mostram resultados dos mesmos tipos de experimentos em grafos aleatórios com conectividades iguais a 3 e $\log n$. Nestes tipos de grafos aleatórios, quanto maior a conectividade, maior é o número de enlaces adicionados, o que contribui para um diâmetro menor. Assim, nos primeiros, a curva ascendente é mais evidente.

Como se pode observar, a latência de diagnóstico situa-se sempre acima do valor

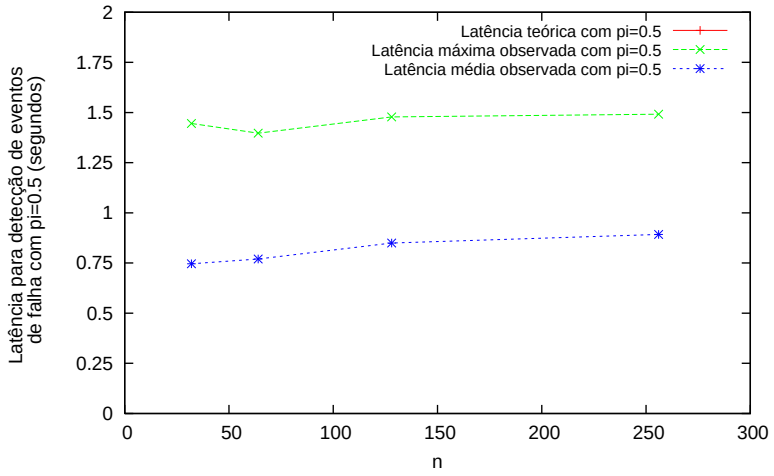

Figura 5.12: Eventos de falha em enlaces - grafos aleatórios $k = 3$ - $\pi = 0,5$ segundos.

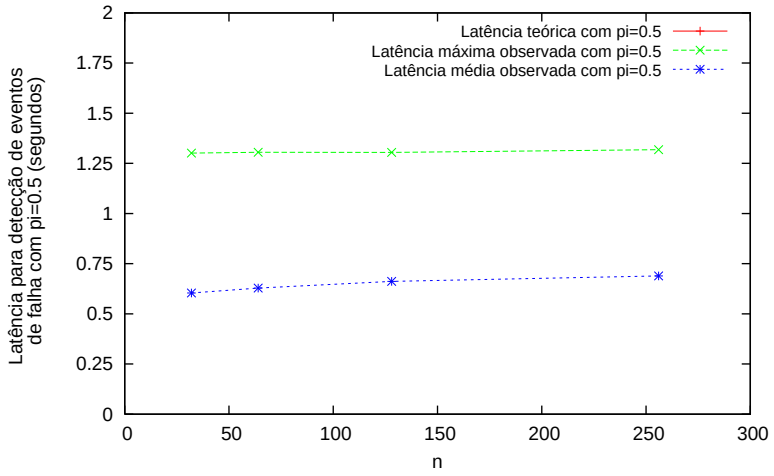

Figura 5.13: Eventos de falha em enlaces - grafos aleatórios $k = \log n$ - $\pi = 0,5$ segundos.

de π, ao contrário dos resultados mostrados nas seções anteriores, o que evidencia a contribuição da latência de disseminação na latência de diagnóstico. A latência teórica não é visível nesta escala.

5.6 NÚMERO DE MENSAGENS UTILIZADAS PELO ALGORITMO

Como a estratégia de disseminação gera mensagens redundantes, o número de mensagens de disseminação é sempre superior ao número de enlaces do componente conexo onde a disseminação ocorre. Entretanto o número de mensagens nunca será superior ao dobro do número de enlaces, uma vez que, se uma mensagem chegar a passar pela segunda vez por um enlace, nunca passará por ele uma terceira vez. Isto ocorre por que o critério utilizado

por um nodo para dar continuidade a uma disseminação é a novidade da informação contida na mensagem, se comparada com a informação contida na tabela de enlaces local.

Este comportamento pode ser observado nas Figuras 5.14 e 5.15. Com experimentos sem particionamento da rede, em que, portanto, o componente conexo é a rede toda, a Figura 5.14 mostra o número médio de mensagens em eventos de falha e de recuperação de nodos em hipercubos. De maneira análoga para experimentos em redes *mesh*, na Figura 5.15.

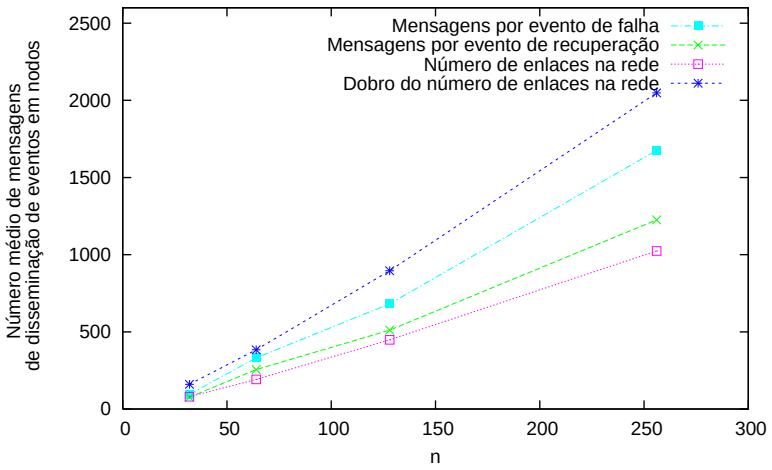

Figura 5.14: Número médio de mensagens - eventos em nodos - hipercubos - média de Poisson 200 segundos.

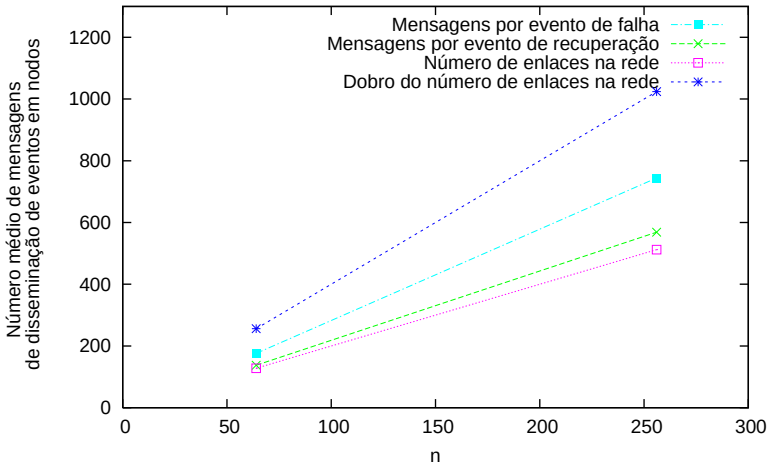

Figura 5.15: Número médio de mensagens - eventos em nodos - redes mesh - média de Poisson 200 segundos.

CAPÍTULO 6

CONCLUSÃO

Este trabalho apresentou o algoritmo *Distributed Network Reachability*. Até a extensão do conhecimento dos autores, este é o primeiro algoritmo de diagnóstico distribuído em nível de sistema concebido para redes particionáveis de topologia arbitrária que executa na presença de particionamentos e *healings* da rede. Além disso, o algoritmo oferece um tratamento formal ao dinamismo de ocorrência de eventos.

Alguns dos principais modelos e algoritmos de diagnóstico em nível de sistema foram apresentados. O trabalho propôs um novo modelo de diagnóstico para tratar a alcançabilidade da rede, no qual novos estados foram definidos (para nodos, *sem-falha* e *inatingível*; para enlaces, *sem-falha*, *não-respondendo* e *inatingível*), os quais formam a base para diagnóstico de redes particionáveis.

O algoritmo foi formalmente especificado em três fases: teste dos enlaces adjacentes, disseminação de informação sobre novos eventos e cálculo local de alcançabilidade. Provou-se que o algoritmo assegura o número mínimo de testes por enlace, ou um testador por enlace por rodada de testes e possui a melhor latência de diagnóstico, obtida pela disseminação dos eventos pela rede de forma paralela. O algoritmo suporta eventos concorrentes com a disseminação de eventos prévios.

Um segundo conjunto de provas formais de correção e desempenho do algoritmo também foi apresentado, este utilizando o arcabouço de *bounded correctness*. Este formalismo foi estendido para contemplar a possibilidade de ocorrência de eventos em enlaces. Tempos de retenção de estados para nodos e enlaces nos estados falho e sem-falha foram obtidos e provou-se que o algoritmo DNR satisfaz as três propriedades de *bounded correctness*: *latência delimitada de diagnóstico*, *latência delimitada de inicialização* e *acuidade*.

Resultados experimentais foram obtidos utilizando técnicas de simulação de eventos discretos. Um simulador do algoritmo foi construído no qual o dinamismo da ocorrência

de eventos foi introduzido por um processo de Poisson. Resultados de simulação foram apresentados para grafos aleatórios com diferentes conectividades de vértices, grafos aleatórios com distribuição *Power-Law* e grafos regulares dos tipos *mesh* e hipercubo. A latência média de diagnóstico do algoritmo foi medida durante a ocorrência de eventos de falha e eventos de recuperação de nodos e enlaces, em situações com particionamento e sem o particionamento da rede.

Embora ambas as abordagens de provas analíticas tenham sido unânimes em demonstrar que a latência de detecção de eventos do algoritmo DNR é da ordem de dois intervalos de teste no pior caso, os resultados experimentais mostraram que a latência média é de um terço do mesmo intervalo em alguns casos.

Foram apresentados resultados experimentais para comparação com o algoritmo *ForwardHeartbeat*, em situações sem particionamentos da rede e com eventos somente em nodos. Concluiu-se que o algoritmo DNR mostrou resultados melhores ou equivalentes aos resultados observados no algoritmo *ForwardHeartbeat*, isto é, com latências médias da ordem de um terço do intervalo de testes para eventos de falha e da ordem do tempo de espera para recuperação em eventos de recuperação. Experimentos com topologias regulares nas mesmas condições apresentaram latências médias similares.

Como trabalhos futuros, pode-se citar a implementação de uma ferramenta SNMP para monitoração de falhas em redes de longa distância, bem como de redes lógicas de topologia arbitrária. Também pode ser citado como abordagem futura de pesquisa, o desenvolvimento de um novo algoritmo para contemplar redes dinâmicas nas quais o conjunto de unidades do sistema não é fixo, isto é, entradas e saídas de nodos são permitidas, como em redes *peer-to-peer*, redes móveis *ad hoc* e redes de sensores.

PUBLICAÇÕES

As publicações decorrentes do trabalho que resultou nesta tese estão listadas a seguir.

- E. P. Duarte Jr. and A. Weber,
  "Simulation of a Distributed Connectivity Algorithm for General Topology Networks,"
  Anais SBC Workshop de Testes e Tolerância a Falhas, SBC/WTF'02, pp. 97-104, Búzios, Maio 2002.

- E. P. Duarte Jr. and A. Weber,
  "A Distributed Network Connectivity Algorithm,"
  Proc. IEEE 6th Intl. Symp. Autonomous Decentralized Systems, IEEE/ISADS'03, pp.285-292, Pisa, April 2003.

- E. P. Duarte Jr. and A. Weber,
  "Computaçãoo Distribuída de Conectividade de Redes de Topologia Arbitrária,"
  Anais SBC Simp. Bras. Redes de Computadores, SBC/SBRC'03, pp. 665-679, Natal, Maio 2003.

- A. Weber, E. P. Duarte Jr. (Supervisor)
  "Towards Healing Partitioned Networks - A PhD. Thesis Proposal,"
  Proc. 1st Latin American Symposium on Dependable Computing - Workshop de Teses e Dissertações, LADC'03, pp. 43-48, Sao Paulo, October 2003.

- A. Weber, E. P. Duarte Jr., and K. V. O. Fonseca,
  "An Optimal Test Assignment for Monitoring General Topology Networks,"
  Proc. 7th IEEE Latin-American Test Workshop, IEEE/LATW'06, pp. 131-136, Buenos Aires, March 2006.

- A. Weber, A. W. Santos, E. P. Duarte Jr e K. V. O. Fonseca,
  "Simulação de um Algoritmo de Diagnóstico Distribuído para Redes Particionáveis de Topologia Arbitrária,"

Anais SBC Workshop de Testes e Tolerância a Falhas, SBC/WTF'08, a ser publicado, Rio de Janeiro, Maio 2008.

- E. P. Duarte Jr., A. Weber and K. V. O. Fonseca,
  "Distributed Diagnosis of Dynamic Fault and Repair Events in Partitionable General Topology Networks," a ser publicado em periódico internacional.

O estágio de doutorando no exterior ("sandwich") foi realizado de setembro/2005 a janeiro/2006 no Instituto de Ciência e Tecnologia da Informação (ISTI) do CNR (Italian National Research Council), em Pisa, Itália, sob orientação dos professores Piero Maestrini e Stefano Chessa.

REFERÊNCIAS BIBLIOGRÁFICAS

[Aguilera, Chen e Toueg, 1997] M. K. Aguilera, W. Chen, S. Toueg, "Heartbeat: a Timeout-Free Failure Detector for Quiescent Reliable Communication," *Proc. 11th International Workshop on Distributed Algorithms*, Sept. 1997.

[Aguilera, Chen e Toueg, 1999] M. K. Aguilera, W. Chen, S. Toueg, "Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks," *Theoretical Computer Science*, 220(1):3-30, June 1999.

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

[Avizienis, Laprie, Randell *et. al.*, 2004] A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, pp. 11-33, Jan. 2004.

[Ayanoglu, Gitlin e Mazo, 1993] E. Ayanoglu, C. Gitlin, J. Mazo, "Diversity coding for transparent self-healing and fault-tolerant communication networks," *Transactions on Communications*, 41(11):1677-1686, November 1993,

[Babaoglu, Davoli e Montresor, 2001] O. Babaoglu, R. Davoli, A. Montresor, "Group Communication in Partitionable Systems: Specification and Algorithms," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 308-336, 2001.

[Bagchi e Hakimi, 1991] A. Bagchi, S.L. Hakimi, "An Optimal Algorithm for Distributed System-Level Diagnosis," *Proc. 21st Fault Tolerant Computing Symp.*, Jun., 1991.

[Barsi, Grandoni e Maestrini, 1976] F. Barsi, F. Grandoni, P. Maestrini, "A theory of diagnosability without repair," *IEEE Transactions on Computers*, Vol. C-25, pp. 585-593, Jun., 1976.

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

[Birman, 1993] K. Birman, "The Process Group Approach to Reliable Distributed Computing," *Comm. ACM*, vol. 36, no. 12, pp.36-53, Dec. 1993.

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

[Blough, Sullivan e Masson, 1988] D. Blough, G. Sullivan, G. Masson, "Almost Certain Diagnosis for Intermittently Faulty Systems," *Proc. 18th Intl. Symp. on Fault-Tolerant Computing*, pp. 260-265, Tokyo, 1988.

[Blount, 1977] M. Blount, "Probabilistic Treatment of Diagnosis in Digital Systems," *Proc. 7th Intl. Symp. on Fault-Tolerant Computing*, Los Angeles, 1977.

[Bu e Towsley, 2002] T. Bu, D. Towsley, "On Distinguishing between Internet Power Law Topology Generators," *IEEE INFOCOM*, vol. 2, pp 638-647, 2002.

[Caruso, Albini e Maestrini, 2003] A. Caruso, L. Albini, P. Maestrini, "A New Diagnosis Algorithm for Regular Interconnected Structures," *Proc. 1st Latin-American Symp. on Dependable Computing*, pp. 264-281, São Paulo, 2003

[Caruso, Chessa e Maestrini, 2007] A. Caruso, S. Chessa, P. Maestrini, "Worst-Case Diagnosis Completeness in Regular Graphs under the PMC Model," *IEEE Transactions on Computers*, vol. 56, No. 7 pp. 917-924, July 2007.

[Chandra e Toueg, 1996] T. D. Chandra, S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225-267, 1996.

[Chessa e Santi, 2001] S. Chessa, P. Santi, "Comparison-Based System-Level Fault Diagnosis in Ad Hoc Networks," *Proc. 20th IEEE Symp. on Reliable Distributed Systems*, pp.257-266, 2001.

[Chessa e Santi, 2002] S. Chessa, P. Santi, "Crash Faults Identification in Wireless Sensor Networks," *Computer Communications*, Vol. 25, No. 14, Sept. 2002.

[Chlamtac, Conti e Liu, 2003] I. Chlamtac, M. Conti, J. Liu, "Mobile Ad Hoc Networking: Imperatives and Challenges," *Ad Hoc Network Journal*, Vol. 1, No. 1, 2003

[Chockler, Keidar e Vitenberg, 2001] G. V. Chockler, I. Keidar, R. Vitenberg, "Group Communication Specifications: A Comprehensive Study," *ACM Computing Surveys*, Vol. 33, No. 4, pp. 427-469, Dec. 2001.

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

[Cormen, Leiserson, Rivest *et. al.*, 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms,* The MIT Press, 2001.

[Culler e Singh, 1999] D. E. Culler, J. P. Singh (1999) *Parallel Computer Architecture - A Hardware/Software Approach,* Morgan Kaufmann, 1999.

[Dahbura, Sabnani e King, 1987] A. T. Dahbura, K. K. Sabnani, L. L. King, "The Comparison Approach to Multiprocessor Fault Diagnosis," *IEEE Transactions on Computers*, Vol. C-36, No. 3, pp. 373-378, March 1987.

[Défago, Schiper e Urbán, 2004] X. Défago, A. Schiper, P. Urbán, "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," *ACM Computing Surveys*, Vol. 4 pp. 372-421, Dec. 2004.

[Ding, Chen, Xing *et. al.*, 2005] Ding, M., Chen, D., Xing, K., Cheng, X., "Localized Fault-Tolerant Event Boundary Detection in Sensor Networks," *Proc. 24th Annual IEEE Conf. Computer and Communication Societies*, 2005.

[Dolev, Malki e Strong, 1996] D. Dolev, D. Malki, R. Strong, "A framework for partitionable membership service," *Proc. 15th Annual ACM Symp. on Principles of Distributed Computing*, pp. 343-343, ACM Press, 1996.

[Duarte, 1998] E. P. Duarte Jr. (1998) "Um algoritmo para Diagnóstico de Redes de Topologia Arbitrária," *Proc. 1st SBC Workshop on Test and Fault Tolerance*, SBC-WTF'1, pp. 50-55, Porto Alegre, 1998.

[Duarte e Cestari, 2000] E. P. Duarte Jr., J. M. A. P. Cestari, "O Agente Chinês para Diagnóstico de Redes de Topologia Arbitrária," *2nd SBC Workshop on Test and Fault Tolerance*, SBC-WTF'2, pp. 88-93, Curitiba, 2000.

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

[Duarte e Mattos, 2000] E. P. Duarte Jr., G. O. Mattos, "Diagnóstico em Redes de Topologia Arbitrária: Um Algoritmo Baseado em Inundação de Mensagens," *Proc. 2nd SBC Workshop on Test and Fault Tolerance*, SBC-WTF'2, pp. 82-87, Curitiba, 2000.

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

[Duarte e Weber, 2003a] E. P. Duarte Jr., A. Weber, "A Distributed Network Connectivity Algorithm," *Proc. IEEE/ISADS'03*, pp.285-292, Pisa, April 2003.

[Duarte e Weber, 2003b] E. P. Duarte Jr., A. Weber, "Computação Distribuída de Conectividade de Redes de Topologia Arbitrária," *Anais SBC/SBRC'03*, pp. 665-679, Natal, Maio 2003.

[Elhadef, Boukerche e Elkadiki, 2007] Elhadef, M., Boukerche, A., Elkadiki, H., "Self-Diagnosing Wireless Mesh and Ad-Hoc Networks using an Adaptable Comparison-Based Approach," *2nd Intl. Conf. Availability, Reliability and Security*, pp. 983-990.

[Fekete, Lynch e Shvartsman, 2001] A. Fekete, N. Lynch,, A. Shvartsman, "Specifying and Using a Partitionable Group Communication Service," *ACM Transactions on Computer Systems*, vol. 19, no. 2, pp. 171-21 6, 2001

[Fussel e Rangarajan, 1989] D. Fussel, S. Rangarajan, "Probabilistic Diagnosis of Multiprocessor Systems with Arbitrary Connectivity," *Proc. 19th Intl. Symp. Fault-Tolerant Computing*, Chicago, 1989.

[Hakimi e Amin, 1974] S.L. Hakimi, A.T. Amin, "Characterization of connection assignment of diagnosable systems," *IEEE Transactions on Computers*, Vol. C-23, pp. 86-88, Jan. 1974.

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

[Harrington, Presuhn e Wijnem, 1999] D. Harrington, R. Presuhn, B. Wijnem, "An Architecture for Describing SNMP Management Frameworks," *RFC 2571*, IETF, May 1999.

[Hiltunen, 1995] M. Hiltunen, "Membership and System Diagnosis," *Proc. 14th Symp. Reliable Distributed Systems*, pp.208-217, 1995.

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

[Julien e Roman, 2004] Huang, Q., Julien, C., Roman, G. C., "Relying on Safe Distance to Achieve Strong Partitionable Group Membership in Ad Hoc Networks," *IEEE Transactions on Mobile Computing*, Vol. 3, No. 2, pp. 192-205, April-June 2004.

[Johnson, 1994] D. B. Johnson, "Routing in Ad Hoc Networks of Mobile Hosts," *Proc. IEEE Workshop on Mobile Computing Systems and Applications*, December 1994

[Kalogeraki, Gunopulos e Zeinalipour-Yazti, 2002] V. Kalogeraki, D. Gunopulos, D. Zeinalipour-Yazti, "A Local Search Mechanism for Peer-to-Peerr Networks," *Proc. 11th Int. Conference on Information and Knowledge Management*, 2002.

[Khanna, Cheng, Varadharajan *et. alli.*, 2007] Khanna, G., Cheng, M. Y., Varadharajan, P., Bagchi, S., Correia, M. P., Veríssimo, P. J., "Automated Rule-Based Diagnosis through a Distributed Monitor System," *IEEE Transactions on Dependable and Secure Computing*, Vol. 4, No. 4, pp. 266-279, October-December 2007.

[Larrea, Fernández e Arévalo, 2004] M. Larrea, A. Fernández, S. Arévalo, "On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems," *IEEE Transactions on Computers*, Vol. 53, No. 7, pp. 815-828, July, 2004.

[Lee e Choi, 2007] M. H. Lee, Y. H. Choi, "Distributed Diagnosis of Wireless Sensor Networks," *Proc. IEEE Region 10 Conference*, pp.1-4, 2007.

[Lumetta e Médard, 2001] S. S. Lumetta, M. Médard, "Towards a Deeper Understanding of Link Restoration Algorithms for Mesh Networks," *Proc. INFOCOM'01*, 2001

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

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

[Maheshwari e Hakimi, 1976] S. Maheshwari, S. L. Hakimi, "On Models for Diagnosable Systems and Probabilistic Fault Diagnosis," *IEEE Transactions on Computers*, vol. C-25, March, 1976.

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

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

[Nassu, Nanya e Duarte, 2007] B. T. Nassu, Nanya, T., E. P. Duarte Jr., "Topology Discovery in Dynamic and Decentralized Networks with Mobile Agents and Swarm Intelligence," *Proc. 7th Intl. Conf. on Intelligent Systems and Applications*, pp. 685-690, 2007.

[Parameswaran, Susarla e Whinston, 2001] M. Parameswaran, A. Susarla, A. B. Whinston, "P2P Networking: An Information-Sharing Alternative," *IEEE Computer*, Jul. 2001.

[Pelc, 1991] A. Pelc, "Undirected Graph Models for System-Level Fault Diagnosis," *IEEE Transactions on Computers*, vol 40, Nov. 1991.

[Poor, Bowman e Auburn, 2003] R. Poor, C. Bowman, C. B. Auburn, *Self-Healing Networks*, ACM Queue vol. 1, no. 3 - May 2003

[Postel, 1981] J. Postel, "Transmission Control Protocol," *RFC 793*, IETF, September 1981. Disponível em ftp://ftp.rfc-editor.org/in-notes/rfc793.txt, Acessado em Maio 2008.

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

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

[Raynal, 2005] M. Raynal, "A Short Introduction to Failure Detectors for Asynchronous Distributed Systems," *ACM SIGACT News*, Vol. 36, No. 1, pp. 53-70, March 2005

[Rodrigues e Guo, 2000] L. Rodrigues, K. Guo, "Partitionable Light-Weight Groups," *Proc. Intl. Conference on Distributed Computing Systems*, pp. 38-45, 2000.

[Santi e Blough, 2002] P. Santi, D. M. Blough, "An Evaluation of Connectivity in Mobile Wireless Ad Hoc Networks," *Proc. IEEE Intl. Conference on Dependable Systems and Networks*, pp. 89-102, Washington, 2002.

[Schiper e Toueg, 2006] A. Schiper, S. Toueg, "From Set Membership to Group Membership: A Separation of Concerns," *IEEE Transactions on Dependable and Secure Computing*, Vol. 3, No. 1, pp. 2-12, Jan-Mar 2006.

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

[Siqueira, Fabris e Duarte, 2000] J. I. Siqueira, E. Fabris, E. P. Duarte Jr., "A Token Based Testing Strategy for Non-Broadcast Network Diagnosis", *1st IEEE Latin American Test Workshop*, pp. 166-171, Rio de Janeiro, 2000.

[Stahl, Buskens e Bianchini, 1992a] M. Stahl, R. Buskens, R. Bianchini, "Simulation of the Adapt On-Line Diagnosis Algorithm for General Topology Networks," *Proc. IEEE 11th Symp. Reliable Distributed Systems*, October, 1992.

[Stahl, Buskens e Bianchini, 1992b] M. Stahl, R. Buskens, R. Bianchini, "On-Line Diagnosis in General Topology Networks," *Workshop on Fault Tolerant Parallel and Distributed Systems*, July, 1992

[Subbiah e Blough, 2004] A. Subbiah, D. M. Blough, "Distributed Diagnosis in Dynamic Fault Environments," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 5, pp. 453-467, May 2004.

[Temal e Conan, 2004] L. Temal, D. Conan, "Failure, connectivity and disconnection detectors," *Proc. 1st French-speaking Conf. on Mobility and Ubiquity Computing*, pp. 90-97, 2004

[Tipper, Dahlberg, Shin *et. al.*, 2002] D. Tipper, T. Dahlberg, H. Shin, C. Charnsripinyo, "Providing Fault Tolerance in Wireless Access Networks," *IEEE Communications Mag.*, pp. 2-8, Jan 2002.

[Weber, Duarte e Fonseca, 2006] A. Weber, E. P. Duarte Jr., K. V. O. Fonseca, "An Optimal Test Assignment for Monitoring General Topology Networks," *Proc. 7th IEEE Latin-American Test Workshop*, pp. 131-136, Buenos Aires, 2006

[Yang e Tang, 2007] X. Yang, Y. Y. Tang, "Efficient Fault Identification of Diagnosable Systems under the Comparison Model," *IEEE Transaction on Computers*, Vol. 56, No. 12, pp. 1612-1618, Dec. 2007.

[Ziwich, Duarte e Albini, 2005] R. P. Ziwich, E. P. Duarte Jr., L. C. P. Albini, "Distributed Integrity Checking for Systems with Replicated Data," *Proc. 11th IEEE Int. Conf. on Parallel and Dependable Systems*, pp. 363-369, 2005.

APÊNDICE

PROVA DO LEMA 16. Considere um nodo X que se recupera no tempo t. A latência de inicialização S de um algoritmo é o tempo máximo que um nodo que se recupera leva para obter um estado L-válido de cada nodo Y e enlace Y–Z no sistema. L é a latência máxima de diagnóstico do algoritmo, isto é, $L = max(2(1+\rho)\pi + (D+4\rho)\Delta_{send\_init} + (D+2+4\rho)\Delta_{send\_max} - \Delta_{send\_min} + \Theta, 2(1+\rho)\pi + (D+1)\Delta_{send\_init} + (D+2)\Delta_{send\_max} - \Delta_{send\_min} + \Theta)$. Como $S \geq L$ por definição, vamos investigar se um nodo ou enlace que permaneceu em dado estado durante o intervalo $[t+S-L, t+S]$ é de fato diagnosticado como tal no tempo $t+S$. Primeiramente consideremos a hipótese de que $S = L$. Sob esta hipótese, o tempo t é o tempo mais tardio em que um nodo ou enlace pode entrar em um dado estado de forma a permanecer naquele estado durante o intervalo $[t, t+S]$.

Esta prova está organizada em duas partes. Primeiramente consideramos um evento de falha em um enlace, adjacente ou não ao nodo X que se recupera, e então um evento de *healing* em um enlace.

Parte a: Evento de Falha. Considere que o enlace Y–Z se torna *não-respondendo* no tempo t. As seguintes situações serão consideradas: ou um dos nodos, digamos Y, torna-se falho no tempo t, ou o enlace Y–Z se torna falho, ou ambos nodo e enlace se tornam falhos no tempo t. Enquanto um destes casos ocorre, o nodo Z pode estar tanto recuperando no tempo t, ou estar no estado *sem-falha* no tempo t ou o nodo Z também se torna falho no tempo t. Note que a mesma análise é válida para um enlace adjacente X–Y no case de o nodo $Z = X$ estar recuperando ao tempo t. Como na análise da latência do algoritmo, o pior caso de latência de inicialização ocorre após uma communicação ter sido enviada pelo enlace imediatamente antes que ele se torne *não-respondendo.*

Inicialmente considere que o nodo Y falha no tempo t após enviar uma requisição de teste iniciada no intervalo $(t-(\Delta_{send\_init}+\Delta_{send\_max}), t-\Delta_{send\_init}]$. Se o enlace está sem-falha, a requisição de teste chega ao nodo Z no intervalo $[t, t+\Delta_{send\_max}]$. Se o nodo Z entrou no estado *sem-falha* no intervalo $[t-(1+\rho)W+\Delta_{send\_max}, t]$, no tempo em que

a próxima requisição de teste chega, o nodo Z está no *tempo de espera para recuperação* e a requisição de teste é ignorada. No máximo no tempo $t + (1 + \rho)W$, quando o nodo Z finalmente executa um teste, o nodo Y está no intervalo $SHT_{u\_node} + (1 - \rho)W$, portanto o nodo Z detecta um *time-out*. Após no máximo $D(\Delta_{send\_init} + \Delta_{send\_max}) + \Theta$, o nodo X diagnostica o evento de falha. Isto é menos que a latência máxima de inicialização enunciada. No caso de o próprio enlace *Y–Z* se tornar falho no tempo t, então no tempo em que o nodo Z testa, o enlace estará no intervalo SHT_{u_link} e o nodo Z também detecta um *time-out*.

Uma outra situação ocorre no caso de o nodo Z ter completado o intervalo de *tempo de espera para recuperação* (W) e ter enviado requisições de teste no intervalo $(t - 2(\Delta_{send\_init} + \Delta_{send\_max}), t + \Delta_{send\_max})$. Se o nodo Y também envia uma requisição de teste a Z no intervalo $[t-(\Delta_{send\_init}+\Delta_{send\_max}), t-\Delta_{send\_init}]$ e se torna falho no tempo t enquanto o enlace *Y–Z* permanece sem-falha, testes simultâneos podem ocorrer. Se o nodo Z se torna o testador, ele detecta um *time-out* no máximo no tempo $t + \Delta_{send\_max} + 2(1 + 2ro)(\Delta_{send\_init} + \Delta_{send\_max})$ detectando portanto o estado falho do enlace *Y–Z*. Se Z se torna o nodo testado, a requisição de teste enviada pelo nodo Y chega ao nodo Z no intervalo $[t, t+\Delta_{send\_max}]$. Após detectar o teste simultâneo, a resposta enviada pelo nodo Z chega a Y após $(\Delta_{send\_init} + \Delta_{send\_max})$, quando o nodo Y já está falho. Mais tarde, no máximo no tempo $t + \Delta_{send\_max} + (1 + \rho)\pi$, o nodo Z testa novamente e detecta *time-out*, porque naquele instante de tempo, o nodo Y está no intervalo de $SHT_{u\_node} + (1 - \rho)W$. Portanto, a detecção pelo nodo Z e o subsequente diagnóstico do evento de falha pelo nodo X ocorre antes da latência de inicialização enunciado no LEMA. Se, por outro lado, o enlace *Y–Z* estiver falho durante todo o intervalo, testes simultâneos não ocorrem e o nodo Z, como testador, detecta o *time-out* como demonstrado acima.

Agora considere a situação na qual o nodo Z completa W e inicia requisições de teste no intervalo $[t - 2(\Delta_{send\_init} + \Delta_{send\_max}), t - 2\Delta_{send\_init} - \Delta_{send\_min})$, de tal forma que o nodo Y envia uma resposta no intervalo $(t - (\Delta_{send\_init} + \Delta_{send\_max}), t - \Delta_{send\_init}]$. O nodo Y então se torna falho no tempo t e sua resposta ao teste chega ao nodo Z em $[t, t + \Delta_{send\_max}]$, quando um evento do *healing* é detectado se o enlace *Y–Z* está sem-

falha. O nodo Z testa novamente no máximo no intervalo $[t - 2(\Delta_{send\_init} + \Delta_{send\_min}) + 2(1+\rho)\pi, t - 2\Delta_{send\_init} - \Delta_{send\_min} + 2(1+\rho)\pi]$ e detecta um *time-out* após $2(1+2ro)(\Delta_{send\_init} + \Delta_{send\_max})$, no intervalo $[t+2(1+\rho)\pi+4\rho\Delta_{send\_init}+2(1+2\rho)\Delta_{send\_max} - 2\Delta_{send\_min}, t + 2(1+\rho)\pi + 4\rho\Delta_{send\_init} + 2(1+2\rho)\Delta_{send\_max} - \Delta_{send\_min}]$. Observe que este tempo máximo de detecção é igual à latência máxima de detecção obtida no LEMA 10. Se o nodo Y falha no tempo t, no pior caso no tempo $t + SHT_{u\_node} + (1-\rho)W = t+2(1+\rho)\pi+(D-1+4\rho)\Delta_{send\_init}+(D+3+4\rho)\Delta_{send\_max}-3\Delta_{send\_min}+\Theta$ o enlace está ainda no estado *não-respondendo* e isto é depois do *time-out* acima. Portanto, ainda que o nodo Z detecte um evento de *healing* no tempo $t+\Delta_{send\_max}$, ele detecta depois o estado falho do enlace Y–Z e subsequentemente o nodo X diagnostica o evento no máximo no tempo enunciado neste LEMA. Se, por outro lado, o enlace Y–Z falha no tempo t, não ocorre um evento de *healing*, e o nodo Z detecta *time-out* no intervalo $[t, t + \Delta_{send\_max}]$.

No caso de o nodo Z estar no estado *sem-falha* no tempo t, isto é, após uma rodada inicial de testes, então se o enlace Y–Z se torna *não-respondendo* naquele instante de tempo, a latência máxima de diagnóstico deste evento foi obtida no LEMMA 10, a qual é exatamente a latência máxima de inicialização enunciada.

Finalmente, se o nodo Y ou enlace Y–Z ou ambos se tornam falhos no tempo t, enquanto o nodo Z também se torna falho naquele instante, então o tempo mínimo que o enlace permanece *não-respondendo* é $SHT_{u\_node} + (1-\rho)W$, o que corresponde ao *tempo de retenção de estado para nodos no estado falho* do nodo Z mais seu *tempo de espera na recuperação*. Note que o intervalo acima é maior que SHT_{u_link}. Se o nodo Y estiver também falho, os nodos Y e nodo Z ambos permanecem falhos por pelo menos aquele intervalo, de tal forma que as latências obtidas no LEMMA 10 também se aplicam aos outros enlaces adjacentes aos nodos Y e Z. Tão logo todos aqueles enlaces sejam diagnosticados como *não-respondendo*, o enlace Y–Z é diagnosticado como *inatingível*, e isto leva no máximo a latência provada no LEMA 10, a qual é a latência de inicialização enunciada neste LEMA.

Portanto, a latência com que um nodo X recuperando no tempo t adquire informação de diagnóstico sobre um evento de falha de um enlace, ocorrido no máximo no

tempo t, é limitado pela latência de diagnóstico do algoritmo.

Parte b: Evento de healing. Considere um evento de *healing* em um enlace Y–Z no tempo t. Considere que o nodo X também entra no estado *sem-falha* no tempo t. Duas situações são consideradas: tanto o enlace Y–Z está sem-falha durante $[t, t+S]$ enquanto o nodo Y ou o nodo Z ou ambos entram no estado *sem-falha* no tempo t, ou o nodo Y e o nodo Z estão no estado *sem-falha* durante $[t, t+S]$ enquanto o enlace Y–Z recupera ao tempo t. Note que a mesma analise procede para um enlace adjacente X–Y no caso de o nodo $Z = X$ estar recuperando ao tempo t.

No primeiro caso, se o nodo Z está recuperando enquanto o nodo Y está no estado *sem-falha*, então o nodo Y responde à requisição de teste de Z no máximo no tempo $\Delta_{send\_init} + \Delta_{send\_max}$ após $t + (1+\rho)W$. Esta resposta chega ao nodo Z no máximo após $\Delta_{send\_init} + \Delta_{send\_max}$, e uma disseminação é assegurada devido ao intervalo SHT_{w_node} do nodo Z. O enlace é então diagnosticado como *sem-falha* pelo nodo X antes da latência máxima de inicialização enunciada.

Se, por outro lado, o nodo Y também recupera no tempo t então ambos os nodos Z e nodo Y entram no *tempo de espera para recuperação* simultaneamente. Portanto, no tempo $t + (1+\rho)W$ quando eles enviam requisições de teste, testes simultâneos podem ocorrer. De maneira similar, se o nodo Y está no estado *sem-falha* e executa um teste no tempo em que o nodo Z também executa um teste, testes simultâneos também ocorrem. Em ambos os casos, a mesma latência acima procede se o nodo Z se torna o testador. Se o nodo Z se torna o nodo testado, então no máximo no tempo $\Delta_{send\_init} + \Delta_{send\_max}$, após a latência de detecção acima, ele recebe uma mensagem de *healing* do nodo Y e confirma a ocorrência de um evento de *healing*. O diagnóstico subsequente pelo nodo X ocorre antes da latência de inicialização enunciada.

Na segunda situação, se o nodo Y e o nodo Z estão no estado *sem-falha* durante $[t, t+S]$, enquanto o enlace Y–Z recupera no tempo t, então considerando o SHT_{u_link}, o diagnostico deste evento ocorre no máximo no tempo enunciado no LEMA 11, o qual é o mesmo limite enunciado neste LEMA para a latência de inicialização.

Portanto, no tempo $t + L$, um nodo arbitrário X que recupera no tempo t diag-

nostica um evento de *healing* ocorrido no enlace Y–Z no máximo no tempo t.

Tanto o diagnóstico de um estado *não-respondendo* como o diagnóstico de um estado inicial *sem-falha* também procedem se o enlace Y–Z entra naquele estado *antes* do tempo t, se é assumido que ele permanece naquele estado durante o intervalo $[t, t + S]$ inteiro. Se este não for o caso, o enlace pode mudar de estado durante $[t, t+S]$ mas após o correspondente tempo de retenção de estado. Nestes casos, tanto o nodo X diagnostica o estado prévio por intermédio de mensagens de *healing* recebidas de nodos vizinhos durante a inicialização ou com o recebimento de mensagens de disseminação antes do tempo $t+S$.

A hipótese inicial de que $S = L$ fica, portanto, confirmada. □

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGª ELÉTRICA E INFORMÁTICA INDUST

"Um Algoritmo de Diagnóstico Distribuído para Red Particionáveis de Topologia Arbitrária"

por

Andréa Weber

Esta Tese foi apresentada no dia 21 de maio de 2008, como requisito parcial p obtenção do título de DOUTOR EM CIÊNCIAS – Área de Concentração: Telem Aprovada pela Banca Examinadora composta pelos professores:

Prof. Dr. Elias Procópio Duarte Jr.
(Orientador - UFPR)

Profª. Dr. Keiko Verônica Ono Fonsec
(Co-Orientadora - UTFPR)

Prof. Dr. Raimundo José de Araújo Macêdo
(UFBA)

Prof. Dr. Carlos Becker Westphall
(UFSC)

Profª. Drª. Sílvia Regina Vergílio
(UFPR)

Prof. Dr. Emílio Carlos Gomes Will
(UTFPR)

Visto e aprovado para impressão:

p/ **Prof. Dr. Hugo Reuters Schelin**
(Coordenador do CPGEI)

RESUMO:

Este trabalho apresenta um novo algoritmo de diagnóstico distribuído em nível de sistema, *Distributed Network Reachability* (DNR). O algoritmo permite que cada nodo de uma rede particionável de topologia arbitrária determine quais porções da rede estão alcançáveis e inalcançáveis. DNR é o primeiro algoritmo de diagnóstico distribuído que permite a ocorrência de eventos dinâmicos de falha e recuperação de nodos e enlaces, inclusive com partições e *healings* da rede. O estado diagnosticado de um nodo é ou *sem-falha* ou *inatingível*; o estado diagnosticado de um enlace é ou *sem-falha* ou *não-respondendo* ou *inatingível*. O algoritmo consiste de três fases: teste, disseminação e cálculo de alcançabilidade. Durante a fase de testes cada enlace é testado por um de seus nodos adjacentes em intervalos de teste alternados. Após a detecção de um novo evento, o testador inicia a fase de disseminação, na qual a nova informação de diagnóstico é transmitida para os nodos alcançáveis. A cada vez que um novo evento é detectado ou informado, a terceira fase é executada, na qual um algoritmo de conectividade em grafos é empregado para calcular a alcançabilidade da rede. O algoritmo DNR utiliza o número mínimo de testes por enlace por rodada de testes e tem a menor latência possível de diagnóstico, assegurada pela disseminação paralela de eventos. A correção do algoritmo é provada formalmente. Uma prova de correção no arcabouço *bounded correctness* também foi elaborada, incluíndo *latência delimitada de diagnóstico*, *latência delimitada de inicialização* e *acuidade*. Um simulador do algoritmo foi implementado. Experimentos foram executados em diversas topologias incluindo grafos aleatórios (*k-vertex connected* e *Power-Law*) bem como grafos regulares (*meshes* e hipercubos). Extensivos resultados de simulação de eventos dinâmicos de falha e recuperação em nodos e enlaces são apresentados.

PALAVRAS-CHAVE

Diagnóstico Distribuído, Gerência de Redes, Diagnóstico em Nível de Sistema, Falhas Dinâmicas, *Bounded Correctness*.

ÁREA/SUB-ÁREA DE CONHECIMENTO

1.03.00.00 – 7 Ciência da Computação

1.03.04.04 – 5 Teleinformática

2008

Nº:34