JADSON IGOR SIQUEIRA

UMA ESTRATÉGIA PARA DIAGNÓSTICO DISTRIBUÍDO DE REDES DE TOPOLOGIA ARBITRÁRIA

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

Orientador: Prof. Dr. Elias P. Duarte Jr.

CURITIBA
2002

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno *Jadson Igor Siqueira*, avaliamos o trabalho intitulado, ***"Uma Abordagem para Diagnostico de Redes de Topologia Arbitraria"***, cuja defesa foi realizada no dia 30 de agosto de 2002, às quatorze horas, no anfiteatro A do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 30 de agosto de 2002.

Prof. Dr. Elias Procópio Duarte Júnior
DINF/UFPR - Orientador

Profª. Dra. Liane Margarida Rockenbach Tarouco
UFRGS - Membro Externo

Profª. Dra. Cristina Duarte Murta
DINF/UFPR

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

Agradecimentos

Agradeço à minha família que me propiciou o ambiente adequado para que este trabalho pudesse ser realizado. Em especial, agradeço à minha secretária, camareira, cozinheira, motorista, assessora, consultora, amiga e companheira, minha princesa Izabel, que em diversas etapas deste trabalho prestou-me suporte motivacional, emocional e logístico, dosando adequadamente sua presença e seu silêncio.

Agradeço ao meu orientador, Dr. Elias, que conseguiu emular dias com mais de 24 horas e semanas com mais de 7 dias, nas diversas ocasiões em que necessitei de seu auxílio para a realização deste trabalho.

Conteúdo

Resumo

Considere um sistema formado por nodos interconectados por enlaces. Tanto os nodos quanto os enlaces podem assumir o estado falho ou sem-falha. Os algoritmos de diagnóstico distribuído têm como objetivo a identificação dos estados dos componentes de um sistema a partir de seus nodos sem-falha. O algoritmo NBND (*Non-Broadcast Network Diagnosis*) permite o diagnóstico distribuído de eventos em redes de topologia arbitrária. Através de testes periódicos, novos eventos são detectados e posteriormente propagados para os demais nodos. Com base no estado dos enlaces da rede, é possível executar, a partir de qualquer ponto da rede, um algoritmo de conectividade para descobrir quais nodos estão acessíveis e quais estão inacessíveis. O SNMP (*Simple Network Management Protocol*) é o protocolo padrão utilizado na Internet para gerência de redes. O SNMP trabalha com informações organizadas numa estrutura hierárquica chamada MIB (*Management Information Base*). Neste trabalho uma nova versão do algoritmo NBND é proposta e descrita formalmente. Um sistema distribuído de monitoramento de redes com agentes SNMP baseado no algoritmo NBND para diagnóstico de redes de topologia arbitrária é proposto e sua implementação é descrita, incluindo a NBND MIB, utilizada pelos agentes. Este trabalho traz, ainda, resultados experimentais da utilização do sistema de monitoramento na detecção de falhas em enlaces de duas topologias de rede: hipercubo e anel.

Abstract

Consider a system consisting of nodes and links that connect these nodes. Links as well as nodes may be faulty or fault-free. System-level diagnosis algorithms aim at the discovery of the system components states by the fault-free nodes. NBND (Non-Broadcast Network Diagnosis) algorithm allows the distributed diagnosis of events in networks of arbitrary topology. Through periodic tests, new events are detected and then disseminated to the other nodes. At any time, any fault-free node in the system is able to execute a connectivity algorithm to discover, based in the states of the links, which nodes are reachable and which ones are unreachable. The SNMP (Simple Network Management Protocol) is the standard protocol used in Internet to manage networks. SNMP deals with information organized in a hierarchical structure called MIB (Managemet Information Base). In this work a new version of the NBND algorithm is proposed and formally described. A distributed network monitoring system built with SNMP agents based in NBND is proposed and its implementation is described, including the NBND MIB, used by the agents. Additionally, this work shows experimental results of applying the monitoring system in two network topologies: ring and hypercube.

Capítulo 1

Introdução

A Internet tem se expandido rapidamente. Na figura 1.1, Comer [2] apresenta o crescimento dessa rede mundial, com um número crescente de redes e máquinas sob responsabilidade de cada administrador. Estes números sugerem que os administradores devem contar com ferramentas que aumentem a sua produtividade e lhes possibilitem lidar com um parque de máquinas conectadas cada vez maior.

	Número de Redes	Número de Computadores	Número de Usuários	Número de Administradores
1980	10	10^2	10^2	10^0
1990	10^3	10^5	10^6	10^1
2000	10^5	10^7	10^8	10^2

Figura 1.1: Expansão da Internet conectada.

Além da Internet, os mais variados ambientes computacionais estão cada vez mais dependentes da interconexão das unidades de processamento através das redes de computadores. As novas tecnologias de redes de computadores têm proporcionado conexões

de maior capacidade. No entanto problemas básicos de gerência desses ambientes ainda não estão solucionados. Tarefas como a gerência de falhas, de desempenho, de configuração, de contabilização e segurança, representam grande parte dos custos totais dessas estruturas. Estas tarefas constituem a funcionalidade básica dos sistemas de gerência de redes [29].

Um aspecto básico para gerência de sistemas distribuídos de qualquer natureza, é o diagnóstico da conectividade. Antes mesmo de se falar em disponibilidade de serviços, é necessário que as conexões de rede estejam disponíveis. A maioria dos sistemas integrados de gerência de redes atualmente disponíveis adota um modelo centralizado para diagnosticar a conectividade da rede. Neste modelo, um nodo central testa os demais nodos e mantém uma base de dados local sobre o estado de toda a rede. Geralmente este nodo central possui uma interface para um operador humano que observa os eventos ocorridos e toma ações corretivas quando necessário. Alguns sistemas ainda fornecem acessibilidade remota, desta forma, o estado do sistema pode ser consultado a partir de outros nodos. Entretanto, este tipo de sistema tem um sério problema, apresenta um *single point of failure*, ou, único ponto de falha. Se uma falha ocorrer no nodo central, todo o sistema fica sem monitoramento. Os sistemas de diagnóstico distribuído possuem outra abordagem para o problema, sendo descritos a seguir.

1.1 Diagnóstico Distribuído

Os algoritmos para diagnóstico em nível de sistema são utilizados para detectar falhas em sistemas que consistem de nodos interligados por enlaces [8]. Nestes sistemas, os nodos testam uns aos outros em intervalos regulares de tempo a fim de detectar eventos. Um evento de falha ocorre, por exemplo, quando um nodo pára de responder os testes que recebe. Um

evento de recuperação ocorre, por exemplo, quando um nodo que não estava respondendo passa novamente a responder os testes.

O objetivo dos algoritmos para diagnóstico distribuído é fazer com que os nodos que estão sem-falha estejam a par da situação de todos os nodos do sistema, ou seja, quais estão falhos e quais estão sem-falha.

Os algoritmos de diagnóstico distribuído são agrupados em duas grandes famílias. A família de sistemas representáveis por grafos completos, na qual, entre quaisquer dois nodos do sistema sempre existe um enlace. E a família de sistemas de topologia arbitrária, na qual, entre quaisquer dois nodos do sistema pode, ou não, haver um enlace direto. O objeto de estudo deste trabalho é a aplicação dos algoritmos de diagnóstico distribuído a computadores interconectados por uma rede de computadores de topologia arbitrária, a figura 1.2 ilustra uma rede de topologia arbitrária. Tal configuração corresponde à maioria das redes corporativas hoje em uso, como a Internet.

O *tempo de detecção de eventos* e a *latência* de um algoritmo para diagnóstico em nível de sistema são duas das métricas utilizadas para medir a eficiência deste tipo de algoritmo. O tempo de detecção é o tempo que o algoritmo leva para detectar um evento. A latência é o tempo que o algoritmo leva para disseminar a informação a respeito do evento a todos os nodos do sistema. Outro ponto a ser analisado nestes algoritmos é o *número de mensagens* trocadas durante a disseminação.

O NBND (*"Non-Broadcast Network Diagnosis"*), proposto inicialmente em [6] e mais recentemente em [5], é um algoritmo para diagnóstico distribuído de redes de topologia arbitrária. O NBND apresenta a melhor latência possível na disseminação de eventos, proporcional ao diâmetro da rede. Além disso, o algoritmo utiliza um pequeno número de mensagens de tamanho reduzido durante a disseminação.

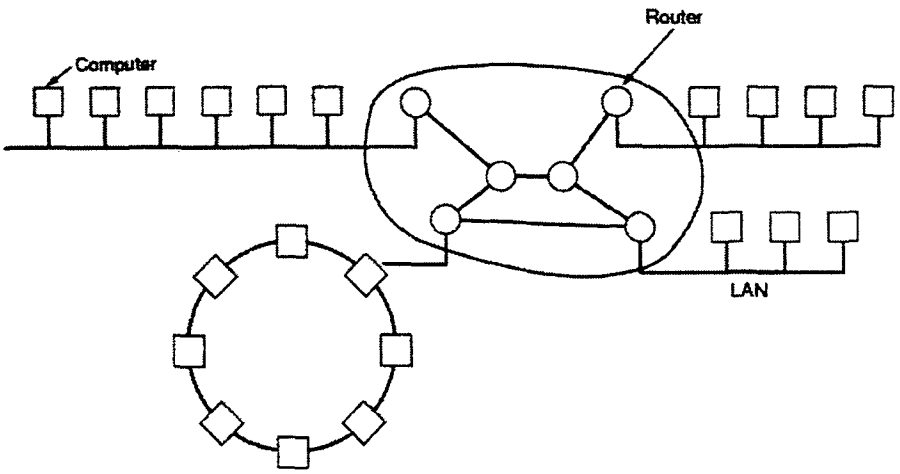


Figura 1.2: Exemplo de rede de topologia arbitrária.

O Algoritmo NBND é dividido em três fases. A *fase de detecção*, na qual novos eventos são descobertos por algum nodo. A *fase de disseminação,* na qual os novos eventos são propagados a todos os nodos. E a *fase de cálculo da conectividade*, na qual um nodo verifica quais nodos estão atingíveis e quais estão inatingíveis.

Este trabalho apresenta uma nova versão do algoritmo NBND, que utiliza na fase de detecção de eventos uma estratégia baseada em passagem de bastão, a *Token-test* [12]. Na disseminação é usada uma estratégia baseada em uma *árvore breadth-first* [7] modificada. Na terceira etapa, um algoritmo clássico de cálculo da conectividade é utilizado para determinar, a partir de qualquer nodo do sistema, quais nodos estão acessíveis e quais estão inacessíveis.

Tais características tornam o NBND uma boa opção para a aplicação prática em sistemas de monitoração de redes de topologia arbitrária. Quando comparado com outros algoritmos para diagnóstico de redes de topologia arbitrária [4, 7], tanto a latência quanto o número de mensagens gasto na disseminação comprovam a superioridade do algoritmo. Em [13] é proposta uma aplicação do algoritmo NBND a *backbones* da Internet.

1.2 Sistemas de Gerência de Redes Baseados em SNMP

O SNMP (*Simple Network Management Protocol*) [10] é o protocolo padrão utilizado para a gerência de redes TCP/IP (*Transmission Control Protocol / Internet Protocol*) [2]. O SNMP é utilizado para gerenciar os mais diversos dispositivos de rede.

O modelo tradicional de gerenciamento com SNMP é chamado modelo gerente-agente [10]. Neste modelo existem *agentes SNMP*, que são processos executados em dispositivos de rede e mantém uma base de informações de gerência. Existem também *estações de gerenciamento*, as estações são responsáveis por coletar informações dos agentes. Com base nas informações coletadas, as estações de gerenciamento podem tomar decisões relacionadas à monitoração e controle da rede.

O agente é executado no dispositivo gerenciado mantendo diversas informações sobre o dispositivo. Por exemplo, no caso de um roteador, informações a respeito das tabelas de rotas e estado das interfaces de rede (*up* ou *down*).

Além de monitorar o dispositivo para manter estas informações atualizadas, o agente tem mais duas funções. O agente responde a consultas SNMP, vindas da estação de gerenciamento, informando a respeito de determinada condição do dispositivo gerenciado. E pode também receber comandos SNMP que o instruem a alterar o estado de determinadas informações que mantêm. Ao alterar estas informações, consequentemente, o estado do dispositivo gerenciado também é alterado.

Por exemplo, um agente SNMP de um roteador poderia responder a uma consulta SNMP informando a tabela de rotas do roteador. Ou então, poderia receber um comando para alterar o estado de uma determinada interface de *"up"* para *"down"*. Esta última alteração faria com que a interface fosse desativada.

As informações a respeito dos dispositivos gerenciados que os agentes mantêm são armazenadas em variáveis que recebem o nome de objetos. No SNMP cada variável corresponde a um objeto. Este conjunto de variáveis recebe o nome de MIB (*Management Information Base*), e é especificado através da linguagem ASN.1 (*Abstract Syntax Notation 1*). Uma introdução ao protocolo SNMP pode ser encontrada em [2] e em [15]. Para uma cobertura mais extensiva ver [10] e [20].

Dada sua grande utilização, sua disponibilidade nos mais diversos dispositivos e ambientes, o SNMP é o protocolo escolhido para a implementação de uma ferramenta baseada no algoritmo NBND aplicável para a monitoração de redes de topologia arbitrária, descrita neste trabalho.

O restante deste trabalho está organizado da seguinte forma. O capítulo 2 trás algumas definições de termos sobre diagnóstico em nível de sistema utilizados durante o trabalho. Em seguida, ainda no capítulo 2, é apresentada uma visão geral sobre algoritmos para diagnóstico de redes de topologia arbitrária. O capítulo 3 apresenta uma nova versão do algoritmo NBND, sua descrição formal, o esboço da prova de correção e um comparativo entre suas estratégias de teste. O capítulo 4 descreve a arquitetura de um sistema distribuído de monitoração baseado em agentes SNMP e a base de informações utilizada na implementação SNMP, a NBND MIB, bem como são apresentados resultados experimentais da utilização dos agentes. No capítulo 5 são feitas algumas considerações finais e sugestões de trabalhos futuros.

Capítulo 2

Diagnóstico de Redes de Topologia Arbitrária

Neste capítulo é apresentado o modelo de diagnóstico utilizado ao longo deste trabalho. É apresentada uma visão geral dos algoritmos para diagnóstico em nível de sistema, bem como uma classificação desses algoritmos. Em seguida, é mostrado um histórico sobre os algoritmos para diagnóstico de redes de topologia arbitrária.

2.1 Diagnóstico em Nível de Sistema

O diagnóstico em nível de sistema tem origem no modelo PMC [18]. Neste modelo os nodos testam uns aos outros interconectados em uma rede onde existe uma ligação direta entre todos os nodos. No modelo PMC todos os resultados dos testes são reportados diretamente a um observador central. Assume-se que a rede nunca falha e trabalha-se com falhas de nodos. A principal característica do modelo é que os nodos sem-falha conseguem, por meio de algum tipo de teste, diagnosticar corretamente o estado dos nodos testados. Os

nodos falhos podem responder arbitrariamente aos testes. O modelo PMC vem evoluindo e foram introduzidas variações em algumas de suas características. Nesta seção descrevemos algumas destas características.

O diagnóstico em nível de sistema tem por objetivo determinar o estado das unidades do sistema. O sistema é composto por *nodos* interconectados por *enlaces*. Neste trabalho os enlaces podem também ser chamados de canais de comunicação, bem como o sistema pode ser chamado de rede. Tanto os nodos como os enlaces podem assumir dois estados, *falho* ou *sem-falha*. A mudança de estado de um nodo ou de um enlace constitui um *evento*. Podem ocorrer dois tipos de evento num sistema, numa *falha*, um nodo ou um enlace passa do estado sem-falha para o estado falho, no caso da *recuperação*, um nodo ou um enlace passa do estado falho para o estado sem-falha. O algoritmo deve então permitir que todos os nodos sem-falha do sistema possam determinar quais nodos estão *atingíveis* e quais estão *inatingíveis* a partir dele.

Para realizar o diagnóstico, o algoritmo pode funcionar *centralizado,* rodando em um único nodo denominado monitor, ou *distribuído,* rodando paralelamente em cada nodo. Ainda, o algoritmo pode determinar o estado da rede quando requisitado, funcionando *off-line*, ou pode estar constantemente monitorando a rede, funcionando *on-line*.

Existem duas categorias principais de algoritmos para diagnóstico. Uma categoria realiza diagnóstico em redes que podem ser representadas por um *grafo completo*, isto é existe um enlace direto entre quaisquer dois nodos da rede. A outra categoria admite que a topologia seja arbitrária, quer dizer, entre dois nodos do sistema pode haver ou não um enlace direto. Este trabalho trata de algoritmos deste segundo tipo, os *algoritmos de diagnóstico de redes de topologia arbitrária*.

Nos algoritmos distribuídos de diagnóstico de redes de topologia arbitrária, distinguimos três fases: *fase de detecção de eventos*, *fase de disseminação* e *fase de cálculo da conectividade*.

A *fase de detecção de eventos* é executada em etapas chamadas *rodadas*. Numa rodada, cada nodo testa seus nodos vizinhos. Quando todos os nodos sem-falha tiverem executado seus testes, uma rodada se completou. O tempo entre a ocorrência de uma falha e a detecção desta por algum nodo da rede é chamado de *tempo de detecção*.

Uma vez detectado um evento, inicia-se a *fase de disseminação*, na qual a informação a respeito do novo evento é propagada para todos os nodos do sistema. O tempo necessário para que todos os nodos sem-falha do sistema façam o diagnóstico de um determinado evento é uma métrica de eficiência de um algoritmo para diagnóstico, e é chamado de *latência* do algoritmo [5]. Na fase de disseminação, geralmente os nodos trocam um grande número de mensagens para que a nova informação chegue a todos os nodos. O tráfego adicional gerado na rede por estas mensagens representa o *overhead* do algoritmo na rede, este tráfego adicional é outra métrica de eficiência do algoritmo.

Na *fase de cálculo da conectividade*, um nodo, de posse do grafo da topologia da rede, pode detectar quais nodos estão *acessíveis* e quais estão *inacessíveis* a partir dele. Essa fase pode ser executada a qualquer momento por qualquer nodo sem-falha da rede.

Nesta seção foram definidos alguns termos relacionados a diagnóstico em nível de sistema que serão utilizados no decorrer de todo o trabalho. O restante deste capítulo apresenta um histórico dos algoritmos distribuídos de diagnóstico de redes de topologia arbitrária.

2.2 O Algoritmo de Bagchi e Hakimi

Em [1], Bagchi e Hakimi introduziram um algoritmo para diagnóstico de redes de topologia arbitrária. O algoritmo usa o modelo de falhas PMC elaborado por Preparata Metze e Chien [18]. No modelo PMC os nodos falhos são imprevisíveis, mas os nodos sem-falha são capazes de determinar corretamente o estado de qualquer um de seus vizinhos. O algoritmo de Bagchi e Hakimi considera falha de nodos, e assume que todos os enlaces funcionam corretamente. Para evitar problemas decorrentes de concorrência, considera-se uma rede na qual cada enlace só transmite informação em um sentido a cada instante e as mensagens seguem o comportamento de uma fila FIFO (*First In First Out*). As falhas dos nodos são permanentes e não ocorrem durante a execução do algoritmo.

O algoritmo é distribuído, não existe uma entidade central para escalonamento ou sincronismo das mensagens e não há a necessidade de nenhum nodo manter um estado global do sistema. Inicialmente, cada nodo sabe apenas o seu próprio número de identificação (*id*), e os *id's* de seus vizinhos diretamente conectados. Um nodo u sem-falha começa testando sucessivamente seus vizinhos. Um nodo vizinho testado, v, também começa a testar os seus vizinhos. Desta forma, uma árvore com raiz no nodo u, contendo apenas nodos sem-falha, é formada. Ao mesmo tempo, outros nodos podem começar a formar suas árvores. À medida que as mensagens de árvores diferentes se encontram, as informações a respeito das árvores são combinadas e as árvores são fundidas. Ao final, tem-se uma única árvore que contém todos os nodos sem-falha do sistema.

2.2.1 Estrutura das Mensagens

Conforme descrito pelos autores, o algoritmo utiliza dois tipos de mensagem as *mensagens informativas* e as *mensagens de formação da árvore*. As mensagens de formação da árvore, base do algoritmo, são compostas dos seguintes campos:

- *PKT.id*: contém o identificador do pacote, usado para identificar a árvore. Coincide com o identificador do nodo raiz da árvore.
- *PKT.size*: o tamanho do pacote é dado pelo número de árvores que já foram fundidas para formar a árvore corrente. É inicializado com 1, e é utilizado para comparar 2 árvores durante uma fusão ($PKT(r_i) > PKT(r_j) \rightarrow PKT(r_i).size > PKT(r_j).size$). O identificador da árvore *PKT.id* é utilizado como critério de desempate caso as árvores tenham o mesmo tamanho.
- *PKT.visited*: É uma lista contendo todos os nodos já diagnosticados pela árvore, ou pelo pacote corrente. Existe um campo *faulty* associado aos nodos falhos. *PKT.visited* é utilizado para evitar que um nodo seja diagnosticado duas vezes.
- *PKT.alldone*: Na fase de broadcast, se *PKT.alldone*=1 indica que o pacote contém informações a respeito de todos os nodos do sistema. Na fase de formação da árvore é utilizado por um filho para indicar ao seu pai que todos os seu vizinhos foram visitados.

Quando dois pacotes, tais que $PKT(r_i) > PKT(r_j)$, são fundidos, ocorre o seguinte nos campos do pacote resultante $PKT(r_i^*)$:

$PKT(r_i^*).id \leftarrow PKT(r_i).id$;

$PKT(r_i^*).size \leftarrow PKT(r_i).size + PKT(r_j).size$;

$PKT(r_i^*).visited \leftarrow PKT(r_i).visited \cup PKT(r_j).visited$;

$PKT(r_i^*).alldone$ não é afetado pela fusão, uma vez que é utilizado localmente pelos nodos.

2.2.2 Formação da Árvore

O algoritmo inicia quando um, ou mais nodos, espontaneamente acordam e criam um pacote. Inicialmente, considere que um único pacote $PKT(r_i)$ foi criado pelo nodo u, dando início a árvore T_i. O nodo u escolhe, para testar, um nodo v que pertença ao conjunto de seus vizinhos mas não pertença ao conjunto de nodos já visitados, ou seja, $v \in N(u)$, $v \notin PKT.visited$. Nos testes do algoritmo de Bagchi e Hakimi, quando o nodo u diagnostica v, v é "acordado" e também diagnostica u.

Caso o teste indique que ambos estão sem-falha:

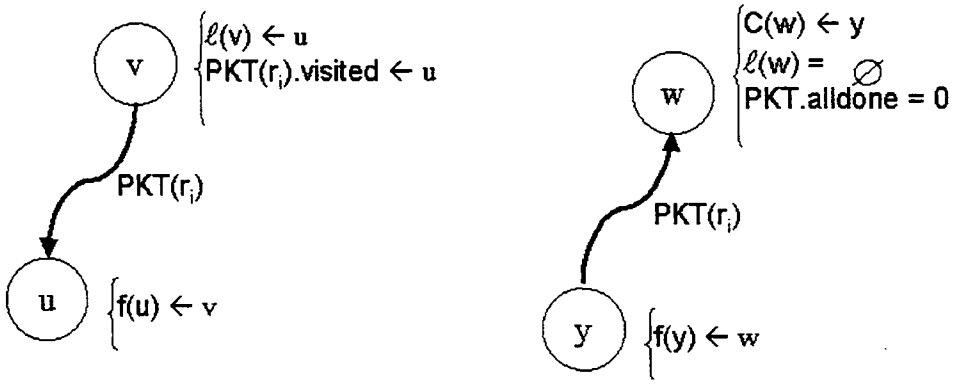
- u inclui v em $PKT(r_i).visited$;
- u manda o pacote para v e guarda v como seu último filho testado: $\ell(v) = u$;
- v guarda u como seu pai: $f(v) \leftarrow u$; e repete o processo selecionando um de seus filhos para o teste.

Caso o teste de u indique que v está falho, u inclui v em $PKT.visited$ como falho e procura um outro nodo para testar. (Seria possível também que u estivesse falho, neste caso v detectaria a falha e ignoraria qualquer pacote recebido de u.)

O processo continua de nodo em nodo, aumentando a árvore T_i, até que um nodo folha seja atingido. O nodo folha é detectado quando: em um nodo y, $N(y) \subseteq PKT(r_i).visited$, ou seja, o estado de todos os vizinhos de y já são conhecidos na árvore T_i.

Neste ponto, y manda o pacote para seu pai com $PKT.alldone$=1, tome-se $f(y) = w$. O pai, w, recebe o pacote e inclui y em seu conjunto de "outros filhos", $C(w) \leftarrow y$. O nodo w seta $\ell(w) = \varnothing$ e $PKT.alldone = 0$. O pai, w, agora possui o pacote e tenta achar outro filho

para incluir em T_i. O retorno dos pacotes ocorre até atingir o nodo raiz quando então a formação da árvore é completada. A figura 2.1 ilustra os dois processos: (a) a inclusão de mais um filho na árvore quando o pacote é mandado do pai para o filho, e (b) o retorno de um pacote do filho para o pai, quando o filho não tem mais a quem propagar o pacote.



(a) Pacote mandado do pai para o filho. **(b)** Retorno do filho para o pai.
Figura 2.1 – Formação da Árvore.

Agora considere que existem dois pacotes, $PKT(r_1)$ e $PKT(r_2)$, na rede, criando respectivamente as árvores T_1 e T_2. As duas árvores são fundidas em uma única árvore da seguinte maneira. Todo nodo u pertencente a T_1 ou T_2 guarda uma cópia da mensagem PKT em uma variável local *current_pkt(u)*. Supondo que $PKT(r_1)$ chegue de $u \in T_1$ para $v \in T_2$, se $PKT(r_1) < PKT(r_2) = current\_pkt(v)$, então $PKT(r_1)$ é colocado em espera no nodo u. Passado algum tempo, quando $PKT(r_2)$ volta para v de $\ell(v)$, o pacote da maior árvore, $PKT(r_2)$, "captura" o da menor árvore, $PKT(r_1)$. Isto é, as duas árvores são fundidas e T_1 torna-se uma sub-árvore de T_2.

Por outro lado, se $PKT(r_1) > PKT(r_2)$ então $PKT(r_1)$ é mandado pelo mesmo caminho de $PKT(r_2)$, ou seja, de v para $\ell(v)$, numa seqüência que os autores chamam de "caçada". Devido à característica FIFO dos enlaces, e aos pacotes não poderem se cruzar em um enlace,

os dois pacotes se encontrarão em algum nodo. Quando do encontro, os dois pacotes são combinados em um único. Neste caso, T_2 torna-se uma sub-árvore de T_1.

2.2.3 Aplicação do Algoritmo

Bagchi e Hakimi mostram que o algoritmo é ótimo em termos do número de mensagens transmitidas para se conhecer o estado de todos os nodos sem-falha do sistema. Porém, além de algumas restrições quanto ao comportamento da rede, o algoritmo possui a limitação de não poder ser executado *on-line*. Outro ponto negativo do algoritmo é o tamanho das mensagens trocadas, que devem carregar informação a respeito dos nodos por onde passam. Em uma rede com número de nodos suficientemente grande, pode haver uma degradação no desempenho da rede decorrente do tamanho das mensagens trocadas pelo algoritmo.

Os autores citam a aplicação do algoritmo no conhecido problema da eleição em sistemas distribuídos. Para esta classe de problemas o número de mensagens necessário para a eleição seria menor do que o de outros algoritmos clássicos conhecidos. Outra possível aplicação do algoritmo de Bagchi e Hakimi é dentro da estratégia do algoritmo NBND, apresentado no capítulo 3. Seria possível executar o algoritmo de Bagchi e Hakimi uma vez, antes da execução do NBND, para estabelecer de forma distribuída uma árvore geradora com todos os nodos da rede. Através desta árvore geradora, a topologia inicial da rede seria distribuída a todos os nodos. Uma vez que a topologia seja conhecida por todos os nodos, inicia-se a execução do NBND para o diagnóstico *on-line* do sistema.

2.3 O Algoritmo Adapt

Em [14], Stahl, Buskens e Bianchini introduziram e apresentaram resultados de simulação do algoritmo "Adapt". Este algoritmo é executado on-line, monitorando continuamente o estado do sistema. Isto é, durante seu funcionamento podem ocorrer falhas e recuperações de nodos.

Periodicamente, os nodos do sistema testam uns aos outros de acordo com um *grafo de testes fortemente conexo*. O *grafo de testes* é um grafo orientado, no sentido do nodo testador para o nodo testado. *Grafo fortemente conexo* é o grafo no qual entre quaisquer dois nodos do sistema existe um *caminho*. Um *caminho* entre um nodo i e um nodo j corresponde a uma seqüência de arestas no qual o nodo testado de cada aresta coincide com o nodo testador da próxima aresta, sendo que a primeira aresta tem como testador o nodo i e a última tem como testado o nodo j.

Quando ocorre um evento, os nodos sem-falha trocam mensagens para se reorganizar de forma a manter, se possível, o grafo de testes conexo. Para evitar que mensagens redundantes sejam processadas, o algoritmo Adapt utiliza um mecanismo de mensagens "datadas" para eliminar as mensagens redundantes.

O algoritmo Adapt constrói um grafo de testes adaptativo nas fases *Search* e *Destroy*. Na fase *Search,* cada nodo procura adicionar novos testes à árvore de testes. Posteriormente, na fase *Destroy*, são eliminados os testes redundantes. O objetivo é construir um grafo de testes capaz de testar todos os nodos gastando o número mínimo de testes.

O algoritmo tem inicio quando da detecção de um evento. Até então, os nodos testam uns aos outros periodicamente e todos contêm o mesmo vetor *Syndromes*. Quando um evento é detectado, o nodo que o detectou inicia a fase *Search*.

2.3.1 Estrutura de Dados

Cada nodo possui um identificador chamado n_i e um vetor denominado *Syndromes*. Cada entrada do vetor *Syndromes* contém uma lista dos nodos testados por um determinado nodo, ou seja, *Syndromes[j]* contém uma lista dos nodos que foram testados pelo nodo n_j. Além do resultado do teste, cada teste do vetor contém também um *timestamp* usado para identificar cronologicamente o teste.

Um nodo n_i recebe informações dos vetores *Syndromes* de outros nodos através de pacotes. Os pacotes são usados para distribuir informações entre nodos e para indicar quando os nodos devem reorganizar seus grafos de testes, ou "adaptar" seus testes.

Os pacotes são divididos conforme as fases do algoritmo, e são de três tipos: *Search*, *Destroy* e *Inform*. Todos os pacotes contêm: o identificador do nodo que originou o teste, uma cópia do vetor *Syndromes* que está armazenada no nodo e uma lista de nodos que já receberam e repassaram o pacote. As fases do algoritmo Adapt são brevemente descritas a seguir.

2.3.2 Fase Search

Na fase *Search*, o nodo checa cada um de seus vizinhos. Um teste para um vizinho é adicionado, caso não exista um caminho para o vizinho no grafo atual. Uma versão do algoritmo de Dijkstra para menor caminho em grafos orientados [19] é executada no vetor *Syndromes* local para se identificar os possíveis caminhos. Em seguida, um pacote *Search* contendo o novo vetor *Syndromes* é gerado e disseminado para todos os nodos sem-falha do sistema.

Cada nodo que recebe o pacote *Search* dispara localmente o mesmo procedimento. O diagnóstico completo do sistema é atingido ao final da fase *Search* quando todos os nodos recebem o vetor *Syndromes* atualizado.

2.3.3 Fase Destroy

Ao final da fase *Search*, possivelmente o grafo de testes contém testes redundantes, na fase *Destroy* estes testes redundantes são eliminados. A fase *Destroy* deve ser executada em cada nodo da seguinte maneira. Um teste para um vizinho é removido, se existir no grafo de testes um outro caminho para este vizinho. Para garantir que o grafo de testes permaneça conexo, o procedimento *Destroy* tem que ser executado seqüencialmente, em um nodo de cada vez.

2.3.4 Fase Inform

Ao término da fase *Destroy* o grafo de testes é um grafo com número de arestas mínimo, porém suficiente para ser um grafo fortemente conexo. Tal grafo é conhecido na literatura de grafos como *minimally strong connected*, neste grafo, o primeiro nodo do caminho que passa por todos os nodos é também o último nodo, ou seja, é formado um ciclo.

Apesar de todos os nodos sem-faha possuírem o estado correto do sistema, eles podem ter armazenado diferentes vetores *Syndromes* durante a fase *Destroy*. Um pacote *Inform* é usado para atualizar o sistema com o vetor *Syndromes* correto. Este pacote é enviado do nodo no qual a fase *Destroy* termina para todos os nodos sem-falha do sistema.

2.3.5 Considerações

O grafo de testes usado no algoritmo Adapt possui a propriedade de ser o menor grafo fortemente conexo cujos nodos são os nodos sem-falha. Contudo para construir o grafo de testes, o algoritmo emprega um procedimento distribuído que requer grandes quantidades de enormes mensagens, provocando possivelmente um grande *overhead* no sistema.

Comparado com o algoritmo de Bagchi e Hakimi, o Adapt possui a vantagem de poder ser executado *on-line*, ou seja, durante a execução do algoritmo pode ocorrer a falha e recuperação de nodos. Entretanto, o tamanho e a quantidade de mensagens trocadas pelo algoritmo Adapt são maiores do que no algoritmo de Bagchi e Hakimi.

2.4 O Algoritmo RDZ

Introduzido em [9] por Rangarajan, Dahbura e Ziegler, o algoritmo RDZ também é baseado no modelo PMC. O RDZ, ao contrário do algoritmo NBND, não considera falhas nos enlaces e sim nos nodos. Durante a execução do algoritmo os nodos podem assumir dois estados: falho ou sem-falha.

O algoritmo RDZ funciona da seguinte maneira. Na fase de detecção, os nodos componentes do sistema testam uns aos outros de forma que cada nodo seja testado por exatamente um nodo. Uma vez que um nodo detecta a falha em um nodo testado, o testador inicia a fase de disseminação do novo evento. O nodo testador do nodo que ficou falho, ao ser notificado do evento pára de testá-lo, assim os nodos falhos não são testados periodicamente para detectar o seu retorno ao sistema. Um nodo falho, quando de sua recuperação, manda

uma mensagem para seus vizinhos até encontrar um nodo sem-falha apto a testá-lo. O nodo vizinho que aceita o pedido torna-se o testador do nodo recém recuperado. Assim que o nodo recém recuperado acha um testador para si, ele manda para todos seus outros vizinhos uma mensagem sobre sua recuperação, e desta forma começa a disseminação do evento de recuperação.

Na fase de disseminação, o nodo testador que detectou o evento de falha, ou o nodo recém recuperado que dissemina sua recuperação mandam uma mensagem a respeito do evento a todos os seus vizinhos. Cada nodo que recebe uma mensagem sobre um evento, ao receber a mensagem de um evento, classifica-a como *oldinfo*, *sameinfo* ou *newinfo*. Quando a mensagem está desatualizada em comparação com as informações que o nodo já detém, a informação é classificada como *oldinfo*, o nodo receptor atualiza o nodo do qual recebeu a mensagem com as informações mais recentes. Quando novas informações são trazidas na mensagem, classificada como *newinfo,* o nodo receptor atualiza suas informações locais e dá seqüência à disseminação propagando a mensagem para todos os seus vizinhos. Quando a mensagem que chega está de acordo com as informações que o nodo já detém, sendo classificadas como *sameinfo*, não é necessário dar continuidade à disseminação nem tampouco atualizar o nodo emissor da mensagem de disseminação recebida. Neste caso a única providência a ser tomada é confirmar o recebimento da mensagem de disseminação para o nodo emissor. A propósito, todas as mensagens de disseminação devem ser confirmadas, desta forma a disseminação funciona também como estratégia de detecção de eventos. Uma vez detectado um evento durante a disseminação uma nova disseminação a respeito deste novo evento é iniciada.

2.4.1 Estruturas de Dados no Algoritmo RDZ

O algoritmo RDZ mantém localmente em cada nodo *j* dois vetores contendo informações sobre o sistema. Um primeiro vetor chamado *eventj[]* é indexado de 1 a *N*, onde *N* é o número de nodos no sistema, e contém os estados mais recentes para o nodo local, sobre o estado de todos os nodos do sistema. Um segundo vetor chamado de *testnbj[]*, também contendo *N* posições, guarda informações que denotam a relação de vizinhança e de teste entre o nodo local e cada um dos outros nodos do sistema. As informações são armazenadas da seguinte maneira:

testnbj[i] = 0 indica que *i* não é vizinho de *j*;

testnbj[i] = 1 indica que *i* é testado por *j*;

testnbj[i] = 2 indica que *i* é o testador de *j*;

testnbj[i] = 3 indica que *i* é vizinho de *j*; e

testnbj[i] = 4 indica que *i* é o testador e ao mesmo tempo é testado por *j*.

2.4.2 Estrutura da Mensagem de Disseminação

A fim de evitar a propagação excessiva de mensagens desnecessárias, o corpo da mensagem de disseminação carrega informações indicando os nodos já percorridos. Desta forma, ao dar seqüência à disseminação, cada nodo checa por quais dentre os seus vizinhos a informação já passou e evita o envio da mensagem redundante. Estas informações estão distribuídas em um vetor chamado *intrapath[]* contendo *N* posições de 1 bit, sendo *N* o número de nodos no sistema. *Intrapath[i]* = 0 indica que a mensagem ainda não passou pelo nodo *i* e *intrapath[i]* = 1 indica que a mensagem já passou pelo nodo *i*.

A mensagem de disseminação contém ainda um outro vetor *msg.event[]*, também com *N* posições, no qual guarda um contador de eventos com o estado mais recente de cada nodo do sistema. Este contador de eventos indica tanto os eventos mais recentes como o estado de cada nodo no sistema. Contadores pares indicam nodo sem-falha e contadores ímpares indicam nodo falho.

Em termos do tamanho da mensagem de disseminação, considerando uma rede com N nodos e contadores de evento de 32 bits tem-se:

Tamanho da Mensagem = N * 32 + N bits.

Para N = 100, tem-se um tamanho de mensagem de 420 bits, ou 52,5 bytes.

2.4.3 A Falha Jellyfish

Conforme citado anteriormente cada nodo é testado por apenas um outro nodo, gerando assim um grafo de testes. O grafo de testes pode ser representado por um grafo orientado, onde cada enlace está direcionado do nodo testador para o nodo testado. A figura 2.2 ilustra um grafo de testes onde o nodo A é o testador dos nodos B e E, B é testador dos nodos A e C, o nodo C é testador de D, os nodos D e E são testadores de nodos dentro dos componentes conexos adjacentes aos nodos em evidência. À medida em que eventos ocorrem e o sistema evolui, a configuração de testes pode se alterar.

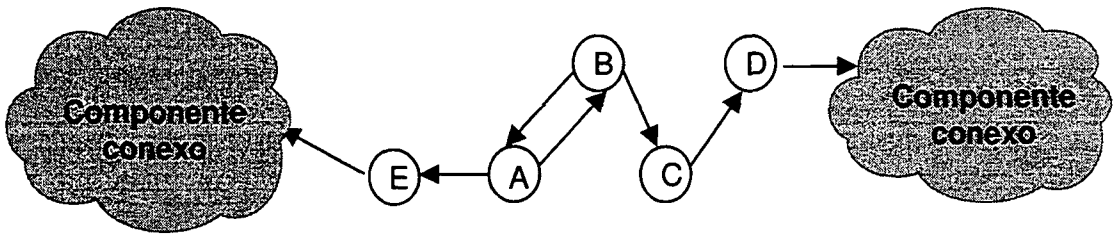


Figura 2.2 – Exemplo de falha *Jellyfish*.

Os próprios autores do RDZ alertam para um caso em que o algoritmo não funciona. Numa configuração especial dos nodos, batizada pelos autores do RDZ de *jellyfish* (água-viva), o algoritmo apresenta problemas na propagação dos eventos. Tal configuração possui uma cabeça e um conjunto de tentáculos. A cabeça consiste de um conjunto de um ou mais nodos que testam uns aos outros em uma configuração cíclica. Os tentáculos são árvores direcionadas de testes que estendem-se a partir da cabeça. A figura 2.2 mostra um exemplo de configuração *jellyfish* conectada a dois componentes conexos. O problema no algoritmo ocorre quando um conjunto de nodos, que constituem uma *jellyfish,* adjacentes a um ou mais componentes conexos falha. Devido ao fato de nenhum dos nodos componentes de uma *jellyfish* ser testado por qualquer nodo do um componente conexo, os componentes não detectam a falha de nenhum dos nodos que constituem a *jellyfish.*

2.5 O Algoritmo de Inundação

Mais tarde, em [4], Duarte e Mattos propõem um algoritmo de diagnóstico de redes de topologia arbitrária baseado em inundação de mensagens. A inundação é uma estratégia bastante simples, mas que tem a vantagem de apresentar a menor latência possível na disseminação. Uma vez detectado um novo evento, o nodo que o detecta manda informações a respeito do evento para todos os nodos adjacentes, estes por sua vez repetem o processo. Qualquer estratégia mais elaborada não pode ser mais rápida do que esta. A desvantagem é que para tanto, um grande número de mensagens redundantes é gerado. Simulações e resultados experimentais são apresentados pelos autores, e os resultados mostram que o

número de mensagens redundantes é, em média, menor que o número máximo possível, que é de duas vezes o número de enlaces.

2.5.1 Descrição do Algoritmo

O algoritmo de diagnóstico baseado em inundação é divido em três etapas: etapa de testes, etapa de disseminação e etapa de diagnóstico. Cada nodo testa seus vizinhos periodicamente e caso seja detectado um novo evento (enlace *sem-falha* se torna *falho*, ou enlace *falho* se torna *sem-falha*), uma mensagem com informações sobre esse evento é disseminada em paralelo para todos seus vizinhos. Cada nodo que recebe a mensagem também dissemina a mensagem aos seus vizinhos, exceto para o nodo que enviou a mensagem, o processo se repete até que todos os nodos sejam informados do novo evento.

O algoritmo baseado em inundação de mensagens considera falha nos enlaces e exige que todos os nodos conheçam a topologia da rede. A cada intervalo de testes, todos os nodos testam todos os enlaces para seus vizinhos. Quando um novo evento é detectado, mensagens são disseminadas notificando os demais nodos da rede da seguinte maneira. O nodo que detectou o evento envia a mensagem para todos os seus vizinhos, os quais, por sua vez, também a enviam para todos os seus vizinhos, exceto para o nodo de onde veio a mensagem, e assim sucessivamente até que a mensagem chegue a todos os nodos do sistema. Este processo de disseminação é chamado de *inundação* ou *flooding* [28].

Assim como no algoritmo RDZ, cada nodo possui um contador de eventos. No começo da execução do algoritmo, o contador é iniciado com zero, indicando o estado sem-falha. A cada novo evento detectado, este contador é incrementado de uma unidade, de forma

que contadores com valores ímpares indicam que o enlace está falho, e contadores com valores pares indicam um estado sem-falha.

A estratégia de inundação não tem controle sobre a seqüência em que as mensagens são distribuídas, podendo ocasionar um grande número de mensagens redundantes, isto é, mensagens que um nodo já recebeu anteriormente.

Considere a topologia de rede representada na figura 2.3, onde ocorre uma falha no enlace entre os nodos A e B. As setas indicam o fluxo das mensagens entre os nodos, supondo que o nodo A detecta a falha no enlace (A-B). A começa a disseminação e envia uma mensagem para todos os seus vizinhos sem-falha, no caso, os nodos C e D. O nodo C ao receber a mensagem de A, envia mensagens para o nodo B e nodo D. Em um instante de tempo muito próximo, o nodo D também recebe a mensagem de A e envia suas mensagens aos nodos C e B.

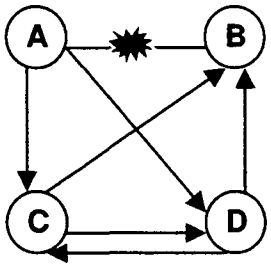


Figura 2.3: Disseminação da falha no enlace (A-B).

Observa-se que, desta forma, os nodos B, C e D recebem a mesma mensagem duas vezes, sendo disseminadas três mensagens redundantes.

É fácil perceber que se as mensagens redundantes não forem tratadas a disseminação continua indefinidamente. Para eliminar as mensagens redundantes, o algoritmo utiliza um mecanismo para descartar tais mensagens. Quando um nodo recebe uma mensagem, ele

compara os contadores das informações contidas na mensagem com os dados que o nodo possui localmente. Se a mensagem recebida não contiver nenhuma informação nova, então o nodo ignora a mensagem. Já se a informação for nova, ele atualiza as informações locais e dissemina a mensagem para seus vizinhos.

A disseminação se completa quando todos os nodos sem-falha recebem as mensagens referentes a um novo estado de um enlace. A latência do algoritmo é a melhor possível, sendo proporcional ao diâmetro da rede.

2.6 O Algoritmo do Agente Chinês

A aplicação do algoritmo do carteiro chinês ao problema de diagnóstico em nível de sistema é proposta em [3]. Neste algoritmo, um agente móvel segue o algoritmo clássico da teoria dos grafos do caminho do carteiro chinês [19]. O agente percorre os nodos para detectar novos eventos. Quando um evento é detectado, o caminho é recalculado e o percurso começa a partir do nodo que detectou o evento e segue por todos os outros nodos disseminando a informação a respeito do novo evento. O novo caminho é utilizado nos diagnósticos subseqüentes até que um novo evento ocorra. Como a disseminação de informações sobre novos eventos é seqüencial, a latência da disseminação é maior do que no algoritmo RDZ e no algoritmo baseado em inundação. Além disso, é feita a asserção de que uma falha não pode tornar o grafo desconexo, situação em geral prevista em algoritmos distribuídos para diagnóstico de redes de topologia arbitrária.

Capítulo 3

O Algoritmo NBND

Neste capítulo, uma nova versão do algoritmo NBND (*Non-Broadcast Network Diagnosis*) é apresentada, é feita uma descrição formal do algoritmo e o esboço da prova de coreção. Este capítulo traz também, um comparativo entre a estratégia de testes tradicional Two-way Test e a nova estratégia Token Test.

O NBND é um algoritmo distribuído, proposto inicialmente em [6], que permite a qualquer nodo sem-falha determinar a conectividade dos nodos e enlaces de uma rede de topologia arbitrária, monitorando continuamente o sistema. Usando o menor número possível de testes, e utilizando um baixo número de pequenas mensagens de disseminação, o algoritmo permite fazer o diagnóstico de conectividade do sistema. Além disto o algoritmo apresenta a menor *latência* possível durante a disseminação, proporcional ao diâmetro da rede [5]. *Latência*, assim como outros termos relacionados a algoritmos para diagnóstico em nível de sistema, utilizados neste capítulo, foram definidos no capítulo 2.

3.1 Descrição Geral do Algoritmo

Considere um sistema composto por *nodos* interconectados por *enlaces*. Neste sistema podem ocorrer falhas nos nodos e nos enlaces. O NBND é um algoritmo distribuído que é executado em cada nodo a fim de identificar as falhas que ocorrem no sistema.

O sistema corresponde a uma rede ponto-a-ponto, ou seja, entre quaisquer dois nodos do sistema pode ou não haver um enlace. Quando não há um enlace direto entre dois nodos é possível que estes se comuniquem passando suas mensagens através de outros nodos. Os enlaces podem assumir dois estados, *falho* ou *sem-falha*. Cada nodo, de acordo com o estado de seus enlaces adjacentes pode apresentar-se no estado *sem-falha* ou *inatingível*.

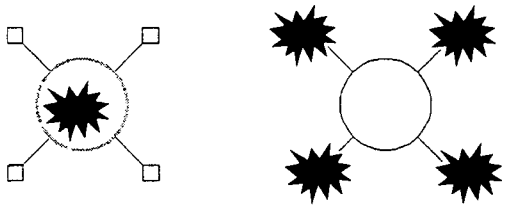


Figura 3.1: Falha no nodo; e falha em todos os enlaces adjacentes ao nodo.

Apesar de poderem ocorrer falhas tanto nos enlaces quanto nos nodos, o algoritmo NBND trata apenas eventos de enlaces. Conforme a "ambigüidade de falhas" [6], ilustrada pela figura 3.1, é impossível para um nodo *i* diferenciar quando um nodo *j* está falho de quando todos os enlaces para o nodo *j* estão falhos. Assim no algoritmo NBND, os nodos trocam mensagens a fim de detectar eventos nos enlaces. Um *evento de falha* ocorre quando um enlace passa do estado sem-falha para falho, e um *evento de recuperação* ocorre quando um enlace passa do estado falho para o estado sem-falha. Após a detecção dos eventos de enlace, com base no estado destes, é possível detectar se determinado nodo está ou sem-falha ou inatingível.

O algoritmo NBND baseia-se no modelo PMC [18] cuja principal característica é que os nodos sem-falha conseguem, através da execução de testes, determinar corretamente os estados dos componentes testados. O procedimento de teste depende da tecnologia do sistema, uma possível técnica de teste é a troca de mensagens entre os nodos, num esquema de *time-out* semelhante ao usado no popular aplicativo *ping* (*ICMP echo request* e *ICMP echo reply*) [2].

Os testes do algoritmo NBND são realizados em intervalos fixos de tempo chamados de *intervalos de testes*. A cada intervalo de testes ocorre uma *rodada de testes*. Numa rodada de testes, cada nodo testa seus enlaces adjacentes de acordo com um *grafo de testes*. Um grafo de testes é um grafo orientado na direção do nodo testador para o nodo testado. O grafo de testes do algoritmo NBND é organizado de forma que um teste é realizado em cada enlace do sistema a cada rodada de testes.

Uma vez que um evento é detectado, mensagens de disseminação a respeito do novo evento percorrem o sistema em paralelo. Uma asserção importante feita pelo algoritmo NBND é que entre a ocorrência de um evento e o término da disseminação deste para todos os nodos do sistema, não podem ocorrer novos eventos.

Cada nodo mantém localmente uma tabela com os contadores de eventos de cada enlace existente na topologia. Estes contadores permitem armazenar o estado do enlace e, durante a disseminação, evitam o repasse de eventos já conhecidos. Cada contador é iniciado com zero, incrementado a cada novo evento detectado e atualizado a cada novo evento recebido. Deste modo, valores pares indicam que o enlace está sem-falha enquanto valores ímpares indicam enlace falho. Com base na tabela de enlaces mantida localmente, cada nodo pode, a qualquer instante, executar um algoritmo de conectividade para determinar quais nodos estão *acessíveis* e quais estão *inacessíveis* a partir dele.

A operação do algoritmo NBND é dividida em três fases:

- *Fase de Testes:* Nesta fase os nodos conectados por um enlace executam um procedimento de teste para detectar eventos nos enlaces.
- *Fase de Disseminação:* Nesta fase um novo evento detectado é propagado para todos os nodos do sistema.
- *Fase de Diagnóstico de Conectividade:* Nesta fase, com base no estado de todos os enlaces da rede, um nodo qualquer pode calcular quais nodos estão acessíveis e quais estão inacessíveis a partir dele.

As fases do algoritmo NBND são descritas nas seções seguintes.

3.2 Fase de Testes

A cada rodada de testes, orientados pelo grafo de testes, os nodos trocam mensagens entre si a fim de detectar eventos nos enlaces. Como o algoritmo NBND segue o modelo PMC, assume-se que os nodos sem-falha são capazes de executar testes com precisão, sempre determinando corretamente o estado dos enlaces testados. Além disso, as informações obtidas são também reportadas com total precisão. Para implementar esta asserção, é necessário determinar um procedimento de teste adequado, muitas vezes dependente da tecnologia do sistema.

No caso de teste de *time-out*, cada nodo manda uma mensagem para o nodo conectado à outra ponta do enlace e inicia um contador de tempo para este teste. Caso a resposta do nodo não chegue depois de excedido o *time-out* configurado, o enlace testado é considerado falho. Há que se notar que o tempo limite de *time-out* pode ser configurado para se adequar à capacidade de transmissão dos enlaces da rede. Além disso, pode ser utilizado um esquema de

"*k* em *j*" mensagens, comumente utilizado em algoritmos de roteamento por estado de enlace como o EGP (*Exterior Gateway Protocol)* e o BGP *(Border Gateway Protocol)* [2].

Para realizar os testes, são apresentadas duas estratégias distintas. As duas diferentes estratégias de testes são chamadas de *Two-way Test* e *Token Test.* Nessa seção apresentamos uma breve descrição das duas estratégias. Na seção 3.5 é apresentado um comparativo entre as duas estratégias.

- A estratégia Two-way Test monitora os enlaces por meio de um teste onde o nodo testador manda uma mensagem para o outro nodo e este, ao receber a mensagem, imediatamente responde com uma mensagem do tipo "eu estou vivo!".

Segue o algoritmo da estratégia Two-way Test:

```
WHILE TRUE
   PARA CADA VIZINHO REPETIR:
   TESTA VIZINHO (MANDA MENSAGEM)
      SE VIZINHO RESPONDER
         SE VIZINHO NÃO ESTAVA RESPONDENDO NO TESTE ANTERIOR
                DISSEMINA MENSAGEM DE NOVO EVENTO ("REPAIR LINK")
         SENÃO
                (OK, VIZINHO CONTINUA VIVO)
      SENÃO (VIZINHO FALHO)
         SE VIZINHO NÃO ESTAVA FALHO NO TESTE ANTERIOR
                DISSEMINA MENSAGEM DE NOVO EVENTO ("FAULTY LINK")
         SENÃO
                (OK, FALHA JÁ ERA CONHECIDA)
```

- Já na estratégia Token Test, o nodo testador manda uma mensagem para o nodo testado, mas a resposta não é imediata, o nodo testador está mandando o *token*, e só deverá recebê-lo de volta ao final de mais um intervalo de testes.

Segue o algoritmo da estratégia Token Test:

```
WHILE TRUE
PARA CADA VIZINHO REPETIR:
SE ESTÁ COM O TOKEN
   MANDA TOKEN (MANDA MSGM PARA VIZINHO)
SENÃO (VIZINHO ESTÁ COM O TOKEN)
   TENTA RECEBER O TOKEN (TIME-OUT)
      SE RECEBEU TOKEN
         SE VIZINHO NÃO ESTAVA RESPONDENDO NO TESTE ANTERIOR
                DISSEMINA MENSAGEM DE NOVO EVENTO ("REPAIR LINK")
         SENÃO
                (OK, VIZINHO CONTINUA VIVO)
      SENÃO (VIZINHO FALHO)
         SE VIZINHO NÃO ESTAVA FALHO NO TESTE ANTERIOR
                DISSEMINA MENSAGEM DE NOVO EVENTO ("FAULTY LINK")
         SENÃO
                (OK, FALHA JÁ ERA CONHECIDA)
```

A nova estratégia de testes, a Token Test, gasta o mesmo número de mensagens que a Two-way Test. Porém ao retardar a resposta, é possível trocar mensagens com um intervalo de tempo que é a metade do tempo usado na Two-way Test. Desta forma, a detecção de falhas em muitos casos é feita num tempo menor do que na Two-way Test.

3.3 Fase de Disseminação

Quando um evento é detectado, inicia-se a fase de disseminação. A disseminação é feita através da construção de uma *árvore de disseminação.* O algoritmo NBND usa uma árvore de busca em largura (*breadth-first tree*) [11] modificada. Esta árvore é construída utilizando os enlaces sem-falha no momento da disseminação e tendo como raiz o nodo que identificou a falha.

A disseminação ocorre da seguinte maneira. O nodo raiz constrói a árvore e manda mensagens de disseminação para seus filhos na árvore. Cada nodo que recebe uma mensagem de disseminação identifica o nodo raiz, e com base neste e na topologia mantida na tabela de enlaces, todos os nodos montam a mesma árvore de disseminação. De posse da árvore de

disseminação cada nodo identifica quais são seus filhos e desta forma a disseminação ocorre paralelamente.

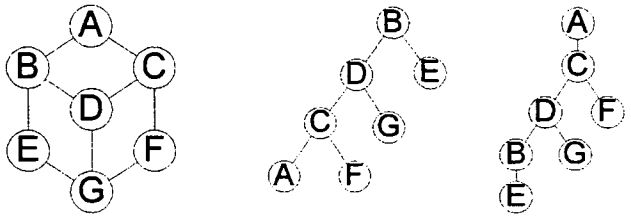


Figura 3.2: Árvores de disseminação para falha entre o nodo A e o nodo B.

A figura 3.2 mostra as árvores *breadth-first* geradas pelo nodo A e pelo nodo B quando da detecção de uma falha ocorrida no enlace entre estes dois nodos, o enlace A-B. Devido à ambigüidade de falhas, o nodo testador, seja ele o nodo A ou o nodo B, não tem como saber se a falha ocorreu no enlace A-B ou no nodo testado. Nota-se pela árvore de disseminação montada pelo nodo A, que se a falha ocorreu no nodo B, o nodo E nunca ficará sabendo dela, mesmo existindo um caminho possível de A para E.

Para corrigir este problema, faz-se uma modificação na árvore *breadth-first* tradicional. A construção da árvore de disseminação é feita de modo que o nodo testado seja colocado o mais próximo possível de um nó folha da árvore [17], conforme apresentado na figura 3.3. No entanto, nem sempre é possível disseminar o novo evento a todos os nodos sem-falha da rede. Quando a falha ocorre num nodo de corte, como ilustrado na figura 3.4, mesmo com a estratégia do nó folha, não há como resolver o problema. Um *nodo de corte* em um grafo conexo é um nodo cuja exclusão do grafo, torna o grafo desconexo. Um *grafo conexo* é o grafo no qual entre quaisquer dois nodos do grafo sempre existe um caminho.

No caso de falha no nodo de corte, a rede será particionada em dois componentes conexos, e em cada componente, os nodos conhecerão apenas o estado atual dos nodos do seu componente conexo.

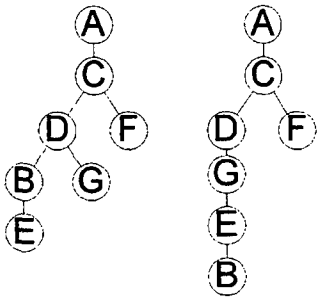


Figura 3.3: Nó B inserido como folha.

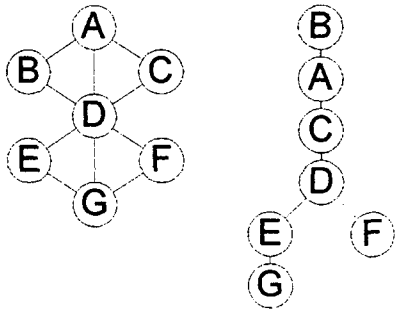


Figura 3.4: Falha no nodo de corte "D".

As mensagens de disseminação possuem três campos, conforme mostra a figura 3.5. Um campo para o nodo que detectou o evento, um para o outro nodo conectado ao enlace testado, e outro campo para o contador de eventos referente ao enlace armazenado no nodo testador. Como as mensagens são pequenas, o impacto delas no tráfego da rede não causa um *overhead* significativo.

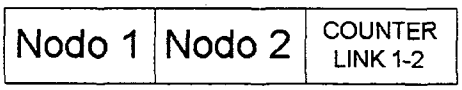


Figura 3.5: Estrutura da mensagem de disseminação.

Ao receber uma mensagem de disseminação, o nodo compara o valor do contador de eventos da mensagem recebida com o valor do contador de eventos para aquele enlace armazenado localmente. Caso o valor local seja maior ou igual ao recebido, o nodo não processa a mensagem, caso contrário, o nodo atualiza o contador de eventos local e dissemina a mensagem. Desta forma evita-se a disseminação de mensagens redundantes.

3.4 Fase de Diagnóstico de Conectividade

Quando requisitado, qualquer nodo pode fazer o diagnóstico da conectividade do sistema. O diagnóstico consiste em determinar se existe um caminho do nodo atual para qualquer outro nodo da rede. Com base no estado de todos os enlaces da rede, mantido localmente por todos os nodos, executa-se um algoritmo clássico de conectividade em grafos [19, 11] para descobrir quais porções da rede estão acessíveis e quais estão inacessíveis a determinado nodo.

3.5 Comparação entre as Estratégias de Testes

Nesta seção, compara-se o desempenho das duas estratégias de testes do algoritmo NBND de modo a ressaltar as características de cada uma e determinar sua eficiência.

Uma métrica utilizada na análise de algoritmos de diagnóstico em nível de sistema é o tempo que o algoritmo leva para propagar um evento (*latência*). Outra métrica importante é o tempo necessário para um evento ser detectado por algum nodo do sistema (*tempo de detecção*). É desejável que o algoritmo detecte um evento na rede o mais rápido possível. Porém, ao desejo de conhecer o estado da rede a todo instante, opõe-se o desejo de que o algoritmo gere baixo *overhead* na rede.

A rapidez com que os eventos são detectados pode ser regulada alterando-se o intervalo de testes. O balanço entre o tempo necessário para a detecção de um novo evento e a quantidade de mensagens que o algoritmo gasta na detecção constitui um *trade-off* importante na eficiência do algoritmo. Considerando esse *trade-off* observa-se que:

- Um menor intervalo de testes proporciona uma detecção de eventos mais rápida, porém gerando mais tráfego;
- Um maior intervalo de testes gera menos tráfego na rede, porém os eventos levam mais tempo para serem detectados.

Por exemplo, se um algoritmo gasta 10 mensagens por unidade de tempo para testar um enlace, detecta um evento mais rapidamente do que se gastasse 2 mensagens por unidade de tempo.

Portanto, para compararmos de forma justa as duas estratégias de testes quanto ao tempo de detecção de eventos, fixamos o mesmo intervalo de testes para as duas estratégias de forma que gastem o mesmo número de mensagens por unidade de tempo.

A figura 3.6 mostra as mensagens utilizadas pelas duas estratégias durante a fase de testes. As duas estratégias de testes garantem que cada nodo fica sabendo do estado de todos os enlaces aos quais estão conectados a cada intervalo de testes, pois neste período ocorre uma rodada de testes completa. No caso da figura 3.6, usando um intervalo de testes de 30 unidades de tempo, nota-se que a estratégia Token Test gasta 7 mensagens num intervalo de 90 unidades de tempo, enquanto que a estratégia Two-way Test gasta 8 mensagens neste mesmo intervalo. Esta diferença de 1 mensagem se mantém seja qual for o tamanho do intervalo analisado, assim pode-se dizer que, para um mesmo intervalo de testes, as duas estratégias gastam o mesmo número de mensagens de teste durante a detecção de eventos.

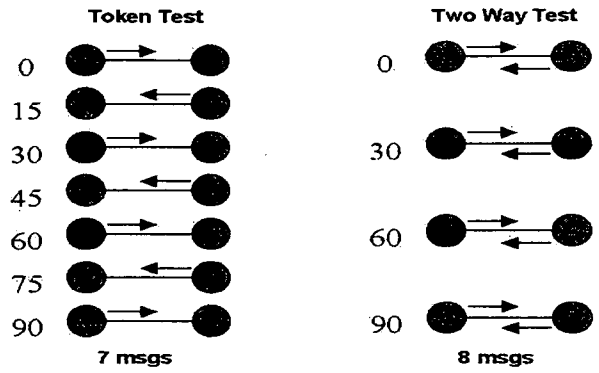


Figura 3.6: Mensagens de detecção de eventos nas duas estratégias de testes.

Conforme o exposto, nota-se que o tempo máximo de detecção de eventos e o número de mensagens utilizadas na detecção são os mesmos para as duas estratégias, porém, para alguns casos, a estratégia Token Test detecta o evento mais rapidamente do que a estratégia Two-way Test. A Token Test consegue isto, pois sua estratégia de testes faz com que mensagens sejam trocadas entre nodos no meio do intervalo de testes. De fato, pode ser mostrado que, na média, a estratégia Token Test é aproximadamente 50% mais rápida na detecção de eventos do que a estratégia Two-way Test.

Podemos expressar este resultado da seguinte maneira:

$$T_{TT} = \frac{1}{2}\, T_{2WT}$$

Onde T_{TT} é o tempo médio de detecção do evento no Token Test e T_{2WT} é o tempo médio de detecção do evento no Two-way Test. O tempo médio de detecção de eventos é dado pela média aritmética dos tempos de detecção observados em diversos instantes de tempo distintos.

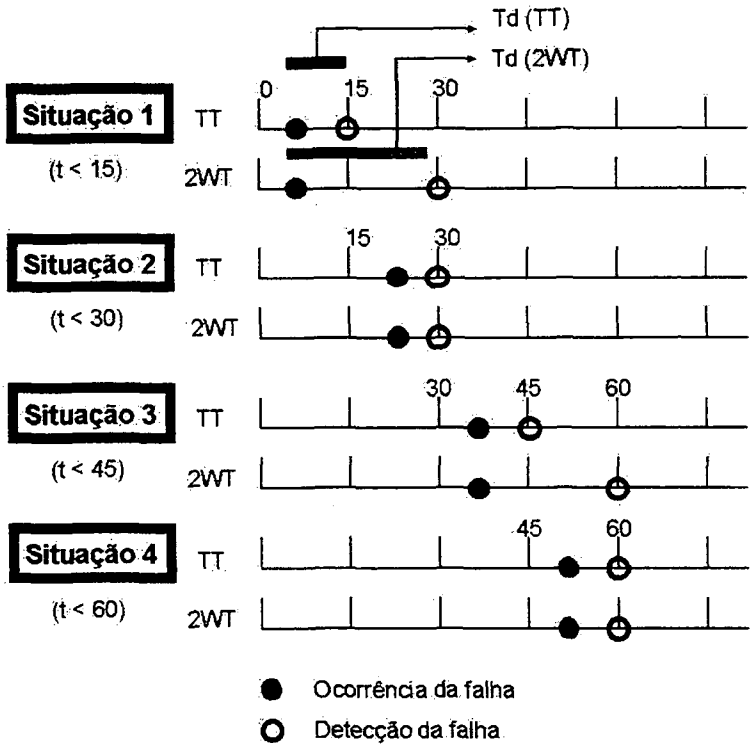


Figura 3.7: Instantes de detecção de falha nas duas estratégias.

A figura 3.7 ilustra a execução das duas estratégias de testes com intervalo de testes (r) igual a 30 unidades de tempo. São observadas ocorrências de falhas no intervalo de tempo de 0 a 60 unidades de tempo. Esse intervalo é grande o suficiente para mostrar todas as possíveis situações de detecção de eventos, após 60 unidades de tempo o processo começa a se repetir, como pode ser observado na figura 3.6.

Na figura 3.7, o processo está dividido em 4 etapas significativas, limitadas pelos momentos onde há troca de mensagens por parte de alguma das duas estratégias. Na primeira situação (SI), ocorre uma falha entre 0 e 15 unidades de tempo. A falha é detectada pela estratégia Token Test no instante de tempo 15, e pela estratégia Two-way Test no instante 30, ou seja, na situação 1, o tempo de detecção da falha na Two-way Test (Td_{2WT}) é igual ao

tempo de detecção da Token Test (Td_{TT}) somado o tempo de meio intervalo de testes (r). Ou seja:

- Em S1: $Td_{2WT} = Td_{TT} + r/2$ (1)

Em *S2*, uma falha ocorrida entre os instantes de tempo 15 e 30, é detectada pelas duas estratégias de teste no instante 30. Ou seja:

- Em S2: $Td_{2WT} = Td_{TT}$ (2)

De forma semelhante, para as situações *S3* e *S4* temos:

- Em S3: $Td_{2WT} = Td_{TT} + r/2$ (3)
- Em S4: $Td_{2WT} = Td_{TT}$ (4)

Conforme observado em (1), (2), (3) e (4) as situações *S1* e *S2* são análogas às situações *S3* e *S4*. Portanto para se analisar todas as possíveis situações de ocorrência de eventos, é suficiente analisar as situações *S1* e *S2*. De (1) e (2) tem-se que para a metade das situações o tempo de detecção de falhas nas duas estratégias é o mesmo, e em outra metade, as falhas são detectadas com meio intervalo de teste de antecedência pela estratégia Token Test.

Note-se que, dentro de cada situação, o tempo de detecção (Td) decresce linearmente conforme aumenta o instante de ocorrência da falha, o que equivale a dizer que o tempo de detecção varia como uma progressão aritmética de razão negativa. Assim, considerando que a probabilidade de ocorrência de uma falha em quaisquer desses instantes dentro do intervalo de cada situação é a mesma, pode-se dizer que o tempo médio de detecção (T) em cada situação é dado pelo termo médio de uma progressão aritmética, ou seja:

$$T = (Td_{inicial} + Td_{final}) / 2 \quad (5)$$

Observa-se na figura 3.7 que o tempo de detecção da estratégia Token Test varia de *r*/2 a 0 em todas as situações. O tempo de detecção da estratégia Two-way Test varia de *r/2* a 0 na situação *S2*, e varia de *r* a *r/2* na situação *S1*. Aplicando (5) a estas observações tem-se:

- Em S1,

  Td_{TT} varia de r/2 a 0, portanto:

  $T_{TT} = (r/2 + 0) / 2 = r/4$ (6)

  Td_{2WT} varia de *r*/2 a *r*, portanto:

  $T_{2WT} = (r/2 + r) / 2 = 3r/4$ (7)

- Em S2,

  $Td_{TT} = Td_{2WT}$, e varia de *r*/2 a 0, portanto:

  $T_{TT} = T_{2WT} = (0 + r/2) / 2 = r/4$ (8)

Como a ocorrência de falhas é distribuída de forma uniforme também entre *S1* e *S2*, de (6),(7) e (8), tem-se que:

- Para todo o intervalo,

  $T_{TT} = (r/4 + r/4) / 2 = r/4$ (9)

  $T_{2WT} = (3r/4 + r/4) / 2 = r/2$ (10)

  E finalmente:

  $T_{TT} = ½ T_{2WT}$ (11)

Ou seja, o tempo médio de detecção de falhas do Token Test é cerca de 50% do tempo médio de detecção de falhas do Two-way Test. A superioridade do da estratégia Token-Test é demonstrada nos resultados experimentais, apresentados no capítulo 4.

É importante observar que a estratégia Token Test garante que, quando ocorrido um evento, no máximo, após $r/2$ unidades de tempo este evento é detectado por algum nodo do grafo. Porém, tomado um nodo qualquer, só é garantido que este nodo fica sabendo o estado de todos os seus enlaces a cada r unidades de tempo, assim como ocorre no Two-way Test.

3.6 Especificação Formal

O modelo utilizado para especificar o algoritmo NBND é um modelo de máquina de estados semelhante ao usado na especificação do algoritmo RDZ [9]. Cada nodo utiliza uma máquina de estados para cada enlace conectado de forma que um nodo adjacente a três enlaces executa simultaneamente três máquinas de estados. De acordo com o modelo estabelecido, cada nodo da rede pode se encontrar em um dos seguintes estados:

1. **Idle;**
2. **Wait-test;**
3. **Wait-diss;**
4. **Wait-repair;**
5. **Fail.**

No estado Idle, o nodo não aguarda retorno de mensagem alguma, o nodo se encontra entre dois intervalos de testes esperando que chegue a hora de executar novos testes. No estado **Wait-test**, o nodo está aguardando a resposta do nodo vizinho para o qual mandou uma mensagem de teste. No estado **Wait-diss** o nodo está aguardando pela confirmação de que a disseminação do último evento terminou. Um nodo entra no estado **Wait-repair** quando um vizinho que estava falho volta a responder. Neste estado o nodo mandou informações atualizadas para o nodo vizinho em recuperação e espera uma confirmação deste vizinho. O estado **Fail** é o estado em que o nodo se encontra após uma falha. A relação entre os estados é ilustrada na máquina de estados da figura 3.8.

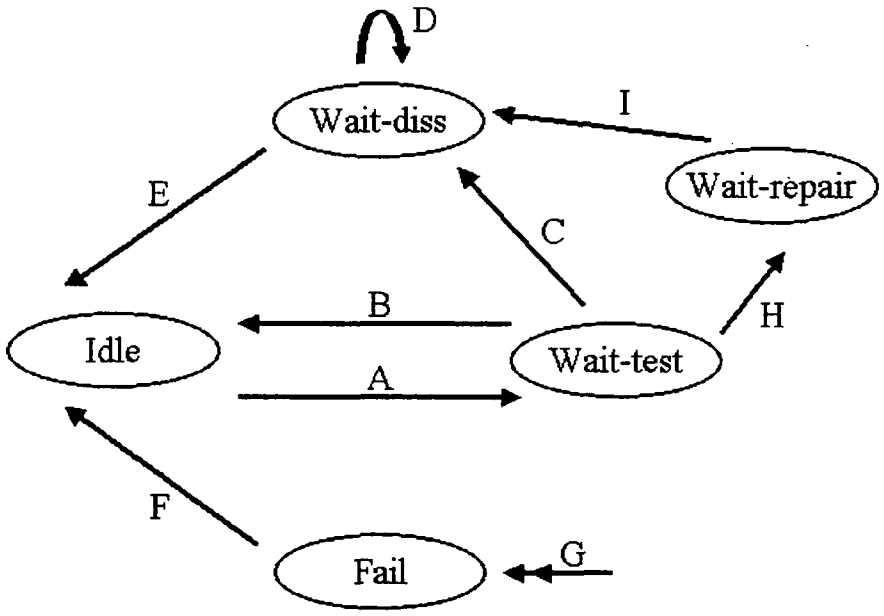


Figura 3.8: Máquina de estados do NBND.

As transições entre os estados são indicadas por letras maiúsculas do alfabeto, e cada transição pode ser realizada por uma ou mais seqüências de eventos. Estes eventos são definidos da seguinte maneira:

- **i!test** : indica o envio de uma mensagem de teste do nodo em questão para o nodo *i*.
- **i?resp** : indica o recebimento de uma resposta do nodo *i* testado. Se o nodo i estava falho, aciona o evento interno *newevent.*
- **?to(resp)**: indica um evento interno no qual o tempo de limite (*timeout*) para o recebimento de resposta se esgotou no nodo em questão. Se o nodo i não estava falho aciona o evento interno *newevent.*
- **bf!diss**: indica que o nodo em questão iniciou uma disseminação por uma árvore *breadth-first.*
- **i?ack(diss)**: indica o recebimento de uma mensagem de confirmação da disseminação vinda de um nodo folha na árvore de disseminação. Caso seja o último nodo folha, aciona o evento interno *enddiss.*
- **i!currinfo:** indica o envio de uma mensagem contendo as informações atualizadas sobre a topologia da rede para o nodo *i*. Neste caso o nodo *i* está em recuperação.

Abaixo segue uma especificação das transições da máquina de estados da figura 3.8, junto ao rótulo da transição está indicado o estado de origem e o estado de destino.

- **A (Idle → Wait-test)**
  to(test) | i!test
- **B (Wait-test → Idle)**
  i?resp | null
  ~newevent

  ?to(resp) | null
  ~newevent
- **C (Wait-test → Wait-diss)**
  i?resp | bf!diss
  newevent | i!currinfo

  ?to(resp) | bf!diss

newevent

- **D (Wait-diss → Wait-diss)**
  i?ack(diss) | null
  ~enddiss

- **E (Wait-diss → Idle)**
  i?ack(diss) | null
  enddiss

- **F (Fail → Idle)**
  i?currinfo | i!ack(currinfo)

- **G (Any → Fail)**
  ?fail | null

- **H (Wait-test → Wait-repair)**
  i?resp | i!currinfo
  newevent

- **I (Wait-repair → Wait-diss)**
  i?ack(currinfo) |null

3.7 Esboço da Prova de Correção

A prova de correção do algoritmo NBND deve levar em consideração a seguinte asserção, um novo evento só ocorre após terminada a disseminação de um evento anterior. Esta asserção é considerada, pois o algoritmo não está projetado para ser usado na presença de falhas intermitentes, onde nodos ou enlaces falham e recuperam-se muito rapidamente.

Define-se um componente conexo sem-falha como sendo o subconjunto máximo de nodos sem-falha para o qual existe um caminho constituído de nodos e enlaces sem-falha entre quaisquer dois nodos sem-falha do componente. Desta forma, por definição, todos os nodos vizinhos de um componente conexo sem-falha estão inacessíveis.

A prova do algoritmo deve contemplar tanto a detecção dos eventos quanto a disseminação das novas informações. Considera-se a ocorrência de uma falha em um enlace, e deve ser mostrado que as afirmações 1 e 2 são verdadeiras:

1- Todo evento é detectado;

2- Após a detecção do evento, e antes da ocorrência do próximo evento, todos os nodos acessíveis dentro do mesmo componente conexo sem-falha têm o mesmo e correto valor de contador para:

2.a) Todos os enlaces dentro do componente;

2.b) Todos os enlaces para nodos vizinhos ao componente.

A afirmação 1, "todo evento é detectado", é sustentada pois o grafo de testes é estabelecido de forma que todos os enlaces são testados a cada rodada de testes, combinando este fato com a asserção "um novo evento só ocorre após terminada a disseminação de um evento anterior" fica fácil perceber que não há como um evento não ser detectado.

Para a afirmação 2, observa-se que durante a disseminação de um evento a árvore de disseminação *breadth-first* modificada contém todos os nodos do componente. Se a falha não particiona a rede então a árvore alcançará todos os nodos do componente informando que um enlace interno ao componente ficou falho. Se a falha particiona a rede, o enlace falho passa a ser um enlace vizinho ao componente, e os nodos que este enlace torna inacessíveis não são incluídos na árvore de disseminação deste componente. Outro ponto a ser observado é que, de acordo com a asserção "um novo evento só ocorre após terminada a disseminação de um evento anterior", durante a disseminação não ocorrem novos eventos, portanto todos os nodos na árvore de disseminação recebem a mensagem acerca do evento.

Capítulo 4

Implementação e Resultados Experimentais

Os sistemas de monitoração de rede disponíveis no mercado são, em geral, do tipo centralizado ou hierárquico. Nestes modelos, um nodo gerente é responsável por testar todas, ou parte, das ligações e determinar a conectividade da rede. Um exemplo de ferramenta baseada em tal modelo é o NetView da Tivoli [16]. No modelo distribuído, a monitoração é executada em vários nodos do sistema, podendo inclusive ser executada em todos eles. O sistema é organizado de forma que cada componente do sistema testa os enlaces conectados ao nodo sobre o qual está sendo executado. Quando necessário, os nodos trocam informações para que todos conheçam o estado global do sistema.

Ambos os modelos têm suas peculiaridades. O modelo centralizado, à primeira vista, parece ser mais facilmente implementado. Porém sua disponibilidade fica comprometida, pois se ocorre uma falha no nodo gerente toda a rede fica sem um monitor. No modelo distribuído, mesmo na presença de falhas em alguns dos nodos, os nodos sem falha continuam monitorando a rede.

Considere que ocorre uma falha na rede, e que esta falha particiona a rede em dois componentes. Neste caso, com o sistema centralizado, a parte da rede que não mantém comunicação com o nodo monitor fica sem informação, mesmo dos nodos do seu componente. No modelo distribuído, dentro de cada componente, cada nodo sem falha continua mantendo informações a respeito dos outros nodos.

Neste capítulo, é descrita a arquitetura de um sistema de agentes SNMP (*Simple Network Management Protocol*) [10] para diagnóstico distribuído de redes de topologia arbitrária, baseado no algoritmo NBND. É mostrada a organização da base de informações que o algoritmo utiliza, a NBND MIB, e a forma como esta MIB é acessada com as diretivas do protocolo SNMP. Em seguida, são apresentados alguns resultados experimentais.

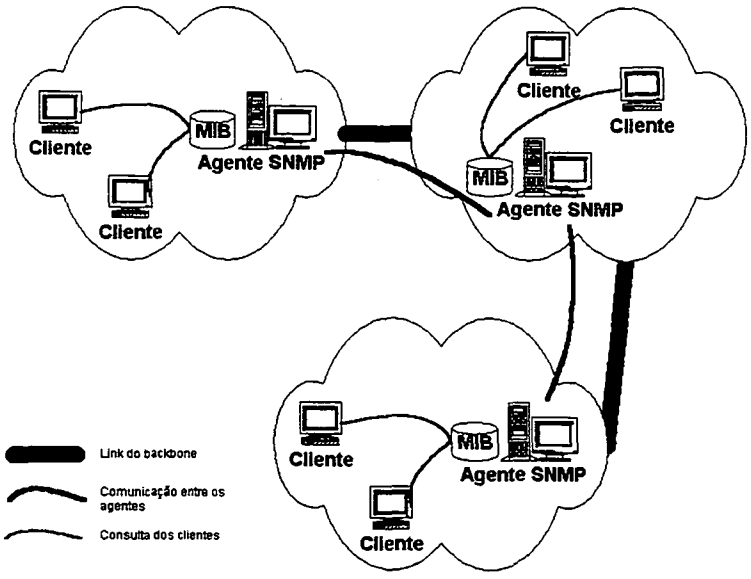


Figura 4.1: Arquitetura do sistema de diagnóstico.

Seguindo o modelo de agentes do SNMP, é proposta uma arquitetura baseada em agentes distribuídos pelos nodos da rede. É possível que o agente esteja presente em cada nodo. Ou ainda, é possível posicionar um agente em cada sub-rede do sistema. O enlace a ser

monitorado não precisa ser um único enlace físico ponto-a-ponto, a conexão lógica monitorada é definida pela relação de vizinhança dos agentes. A figura 4.1 mostra um exemplo de três redes locais conectadas por enlaces de *backbone* onde cada rede local possui um agente monitorando as conexões de *backbone* para outros agentes.

Quando são detectados eventos, os agentes rodando o algoritmo NBND se comunicam de forma a trocar informações a respeito da rede. Os clientes, a qualquer momento, podem consultar os agentes a fim de obter o estado atual do sistema.

4.1 A MIB de Diagnóstico do NBND

O objetivo do algoritmo é disponibilizar aos clientes um mapeamento do estado atual de todos os enlaces da rede. Cada agente SNMP do sistema executa o algoritmo NBND e mantém localmente estes estados em uma MIB do SNMP. Os clientes podem então consultar as MIB mantidas pelos agentes para saber o estado da rede.

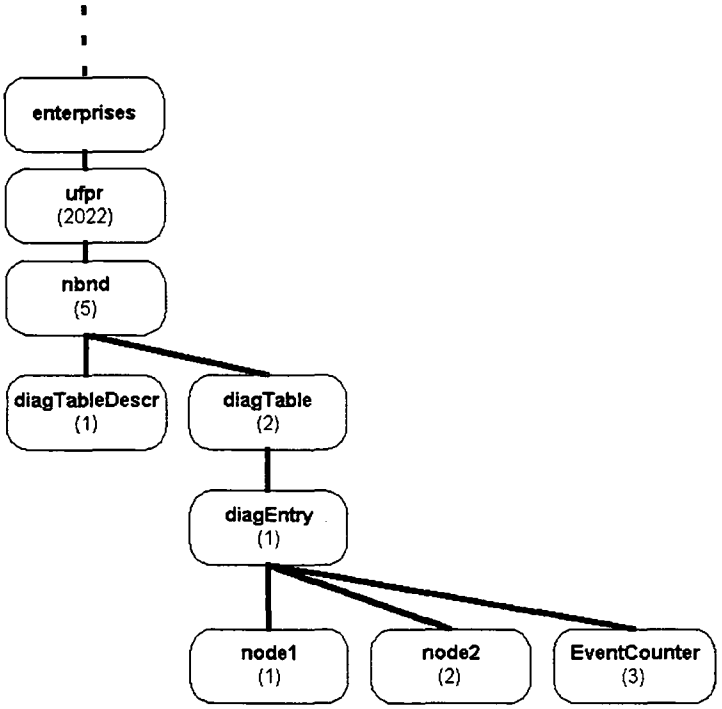


Figura 4.2: Estrutura da NBND MIB.

Nesta seção é proposta a NBND MIB, uma base de dados de informações de gerência do protocolo SNMP desenvolvida para representar a topologia da rede e o estado dos enlaces desta rede. A NBND MIB descreve uma tabela de enlaces da rede com uma entrada para um contador de estados por enlace. Conforme descrito pelo algoritmo NBND, contadores de enlace par representam enlaces sem-falha e contadores ímpares representam enlaces *timing-out*. A figura 4.2 ilustra a árvore da NBND MIB.

As MIBs do SNMP são descritas em ASN.1 *(Abstract Syntax Notation 1)* [10], uma linguagem formal de especificação que contém uma descrição dos objetos gerenciados, os tipos de dados destes objetos e suas propriedades dentro do SNMP. Por esta descrição é possível, também, identificar o ponto onde a MIB descrita é inserida na árvore de MIBs do

SNMP. A MIB do NBND está posicionada dentro do grupo *ufpr*, que por sua vez está dentro do grupo *enterprises*. Abaixo segue a especificação da NBND MIB na linguagem ASN.1.

```
================================================
================================================
NBND-MIB DEFINITIONS ::= BEGIN

-- --  NBND MIB.
-- --
-- --  AUTORES:     ELIAS P. DUARTE JR.
                    JADSON I. SIQUEIRA
-- --
-- --   E-MAIL:     ELIAS@INF.UFPR.BR
                    JADSON@POP-PR.RNP.BR
-- --
-- --  UNIVERSIDADE FEDERAL DO PARANÁ
-- --  DEPARTAMENTO DE INFORMÁTICA, CURITIBA, BRASIL
-- --
-- --  2002
-- --

IMPORTS
       MIB-2, IPADDRESS, COUNTER32, ENTERPRISES, OBJECT-TYPE
          FROM SNMPv2-SMI
       DISPLAYSTRING
          FROM SNMPv2-TC;

UFPR          OBJECT IDENTIFIER ::= { ENTERPRISES 2022 }
NBND          OBJECT IDENTIFIER ::= { UFPR 5 }

DIAGTABLEDESCR OBJECT-TYPE
            SYNTAX  DISPLAYSTRING
            ACCESS  READ-ONLY
            STATUS  MANDATORY
            DESCRIPTION
                  "DESCRIÇÃO DA MIB DE DIAGNÓSTICO DISTRIBUÍDO."
            ::= { NBND 1 }

DIAGTABLE OBJECT-TYPE
        SYNTAX SEQUENCE OF DIAGENTRY
        ACCESS NOT-ACCESSIBLE
        STATUS MANDATORY
        DESCRIPTION
              "UMA TABELA CONTENDO TODOS OS ENLACES DA REDE."
        ::= { NBND  2 }

DIAGENTRY OBJECT-TYPE
          SYNTAX  DIAGENTRY
          ACCESS  NOT-ACCESSIBLE
          STATUS  MANDATORY
          DESCRIPTION
                "CADA ENTRADA CORRESPONDE A UM ENLACE DA REDE.
                O ENLACE É INDEXADO PELO ENDEREÇO IP DO PAR
                DE NODOS QUE O CONECTA (NODO1, NODO2).
                OBS: A TABELA GUARDA ENLACES REPETIDOS,POR EXEMPLO,
                O ENLACE (10.0.0.1, 10.0.0.2) É IGUAL
                AO ENLACE (10.0.0.2, 10.0.0.1)"
          INDEX    { NODO1, NODO2 }
          ::= { DIAGTABLE 1 }

DIAGENTRY ::=
     SEQUENCE {
        NODE1
           IPADDRESS,
```

```
        NODE2
            IPADDRESS,
        EVENTCOUNTER
            COUNTER32
    }

NODE1 OBJECT-TYPE
            SYNTAX  IPADDRESS
            ACCESS  READ-ONLY
            STATUS  MANDATORY
            DESCRIPTION
                "IDENTIFICADOR IP DO PRIMEIRO NODO NO QUAL O ENLACE INCIDE."
            ::= { DIAGENTRY 1 }

NODE2 OBJECT-TYPE
            SYNTAX  IPADDRESS
            ACCESS  READ-ONLY
            STATUS  MANDATORY
            DESCRIPTION
                "IDENTIFICADOR IP DO SEGUNDO NODO NO QUAL O ENLACE INCIDE."
            ::= { DIAGENTRY 2 }

EVENTCOUNTER OBJECT-TYPE
            SYNTAX  COUNTER32
            ACCESS  READ-ONLY
            STATUS  MANDATORY
            DESCRIPTION
                "CONTADOR DE EVENTOS PARA O ENLACE.
                 VALORES PARES INDICAM QUE O NODO ESTÁ "SEM-FALHA",
                 VALORES ÍMPARES INDICAM NODO "TIMING-OUT" "
            ::= { DIAGENTRY 3 }
```

O SNMP provê diretivas de acesso para leitura e para escrita às variáveis das diversas MIBs disponíveis. Os acessos são regidos pelo controle de acesso do SNMP [20]. No SNMP, esse controle pode ser feito com base no endereço IP de origem, numa string de senha chamada *community* e na árvore de MIBs, que pode ser dividida em *visões*. Num ambiente de produção, devem ser tomadas precauções com relação à segurança.

É adequado que os clientes tenham acesso restrito à leitura. Através das visões, pode-se limitar o acesso exclusivamente às variáveis da sub-árvore da NBND MIB. É possível, ainda, restringir esse acesso a endereços IP de clientes previamente estabelecidos.

Os acessos são feitos pelas primitivas GET e SET do SNMP. Abaixo seguem alguns exemplos de acessos e seus significados:

```
GET DIAGTABLE.DIAGENTRY.EVENTCOUNTER.10.0.0.1  . 10.0.0.2
                                    <nodo1>  . <nodo2>
```

- Retorno = 2.

➔ Consulta o eventCounter do enlace entre os nodos 10.0.0.1 e 10.0.0.2. O resultado da consulta retornado é 2. O valor par 2, indica que o enlace está no estado *timing-out*.

```
SET DIAGTABLE.DIAGENTRY.EVENTCOUNTER.10.0.0.1  . 10.0.0.2   =   1
                                    <nodo1>  . <nodo2>    = <eventCounter>
```

➔ Nesse caso um cliente SNMP estaria tentando fazer o identificador do enlace igual a 3, o que representaria uma falha no enlace entre os nodos 10.0.0.1 e 10.0.0.2.

Como são os próprios agentes os responsáveis pela manutenção do estado da tabela de enlaces contida na MIB, os clientes têm acesso somente de leitura às variáveis da NBND MIB, ou seja, podem apenas usar a primitiva GET. Os agentes executam testes periódicos nos nodos vizinhos e à medida que detectam novos eventos atualizam as suas MIBs e disseminam as informações para os outros agentes. Os clientes podem ler as MIBs dos agentes para obter informações a respeito da conectividade da rede.

Os clientes podem consultar na MIB o estado de algum enlace em específico. Podem também consultar a tabela inteira ou uma porção dela, e a partir da tabela, executar um algoritmo de conectividade para descobrir quais nodos estão acessíveis e quais estão inacessíveis a ele.

4.2 O Agente NBND

O *Tcl* (*Tool Comand Language*) [21] é uma linguagem de *scripting* com interpretadores disponíveis em diversos ambientes. O pacote *Scotty* [23], através de sua extensão *Tnm,* provê ao *Tcl* uma interface para comunicação com *sockets* UDP e TCP bem como funcionalidades de cliente e agente SNMP. O *Scotty* fornece um agente SNMP extensível, ou seja, um agente no qual podem ser incorporadas novas MIBs. Com o uso da

linguagem *Tcl* e da biblioteca *Scotty* foi implementada a NBND MIB em um agente SNMP sobre uma pilha de protocolos de comunicação TCP/IP do sistema operacional Linux.

Os experimentos foram realizados sobre o sistema operacional *Conectiva Linux* 7.0 com *kernel* 2.4.5-9cl [26]. Foi utilizada a linguagem *Tcl* 8.3.3 e a biblioteca *Scotty* 2.1.11. Esta seção descreve a implementação e funcionamento do agente NBND.

Para a obtenção de resultados experimentais, vários agentes NBND são executados sobre um mesmo *host*, cada um simulando um nodo de uma rede, desta forma é possível simular diferentes topologias de rede. As topologias são configuradas pelo arquivo *nbnd-conf.tcl*, que contém uma lista dos enlaces que compõem a rede. Os agentes se distinguem uns dos outros pelos números das portas UDP e TCP que utilizam. As mensagens trocadas entre os agentes podem ser identificadas pelos endereços de *loopback* utilizados. A interface *loopback* é definida no protocolo TCP/IP como uma interface virtual para o *host* local. Cada agente usa um endereço de *loopback* diferente, por exemplo, 127.0.0.10 ou 127.0.0.11.

Para simular falhas nos enlaces da rede é utilizado o software de filtragem de pacotes *iptables* versão 1.2.2 [27]. Por exemplo, para simular a falha no enlace entre o nodo 127.0.0.10 e o nodo 127.0.0.11, é aplicado um filtro para bloquear as mensagens com endereço de origem 127.0.0.10 e de destino 127.0.0.11 e um outro filtro que bloqueia o sentido inverso, ou seja, endereço de origem 127.0.0.11 e endereço de destino 127.0.0.10. Desta maneira, os pacotes de teste são bloqueados e o enlace é detectado como falho.

Entretanto, para agentes rodando em um mesmo *host* não é possível especificar o IP de origem das mensagens de detecção de eventos com as ferramentas de detecção utilizadas, o comando *icmp* do *Scotty* e o comando *fping* do sistema Linux. Portanto é necessário usar um esquema alternativo para identificar agentes de origem e agentes de destino nas mensagens. As mensagens de detecção de eventos são mandadas para o endereço

127.0.<Id_de_Origem>.<Id_de_Destino>. Neste esquema uma mensagem para 127.0.10.11 significa um teste do agente em 127.0.0.10 para o agente em 127.0.0.11. Note-se que este esquema de endereçamento limita o numero de *hosts* na topologia da rede a 255 hosts, numero suficiente para os propósitos experimentais.

Para facilitar a configuração dos filtros pode-se usar os *scripts nbnd-filterDown*, *nbnd-filterUp* e *nbnd-filterList* dos apêndices. Por exemplo, um filtro aplicado pelo comando "nbnd-filterDown 10 11" coloca como falho o enlace entre o agente no endereço 127.0.0.10 e o agente no endereço 127.0.0.11. Este filtro é listado pelo *script nbnd-filterList* como:

```
$> nbnd-filterList
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  anywhere             127.0.10.11
DROP       all  --  anywhere             127.0.11.10
```

Os testes dos enlaces são feitos através de um esquema de envio de mensagens e controle do tempo de resposta com *timeout*. Na estratégia de testes Two-way Test, os agentes testam os enlaces que os conectam por meio de mensagens ICMP. No caso da estratégia de testes Token Test, são utilizadas mensagens UDP. Para proceder à disseminação dos eventos detectados são utilizadas conexões TCP entre os agentes.

Cada agente mantém uma instância da NBND MIB contendo os estados de todos os enlaces da rede. Uma vez que um evento seja detectado por algum dos agentes, o novo evento é comunicado aos outros agentes. A qualquer momento um cliente SNMP pode, por meio de diretivas GET do SNMP, fazer consultas a qualquer dos agentes, e obter o estado atual dos enlaces.

A figura 4.3 mostra o aplicativo *mibtree* que fornece uma interface gráfica de cliente SNMP escrita em *Tcl/Tk* [22] com a biblioteca *Scotty*. O aplicativo *mibtree* acompanha o pacote *Scotty* e é utilizado nos experimentos para consultar os agentes e acompanhar o estado da tabela de enlaces da NBND MIB. A janela mais ao fundo é a janela principal do aplicativo *mibtree*. Nesta janela é possível percorrer toda a hierarquia de uma certa MIB e visualizar a descrição e o tipo de cada variável de uma MIB. No exemplo da figura 4.3 é mostrada a hierarquia da NBND MIB.

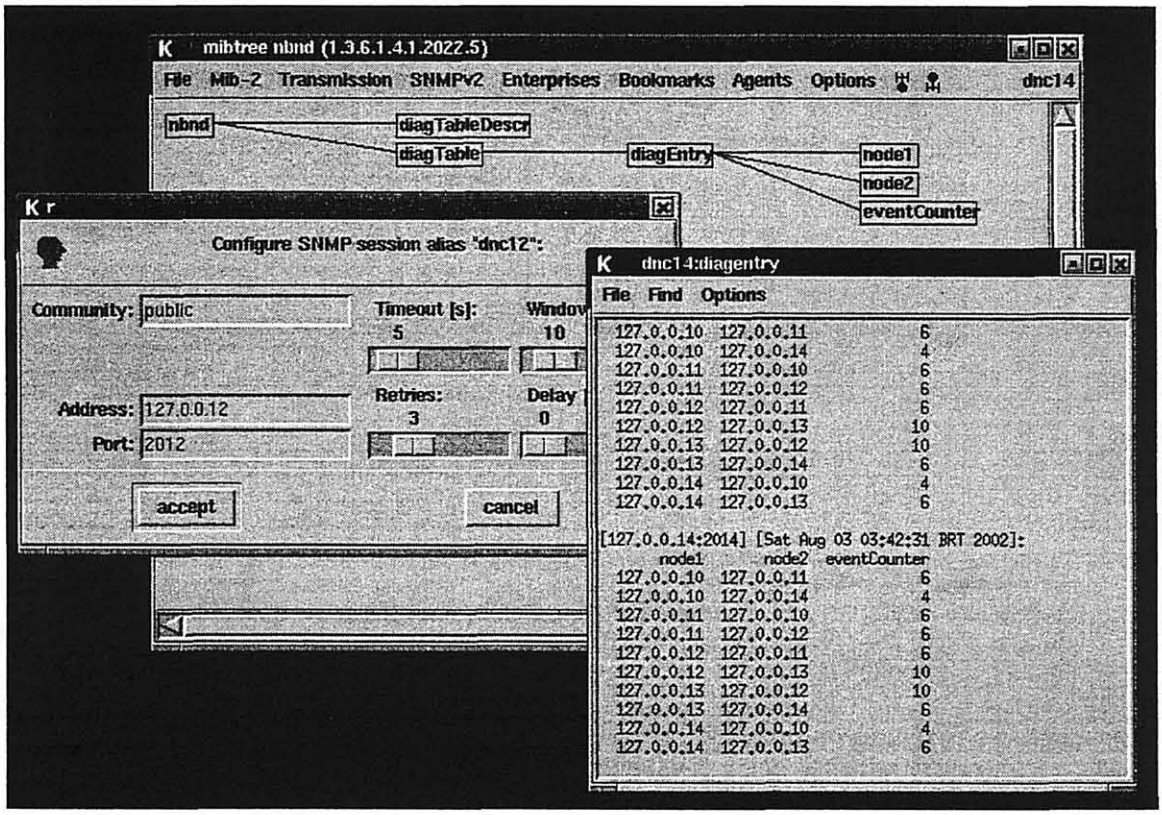


Figura 4.3: Monitoração da NBND MIB com o aplicativo *mibtree*.

A janela do plano intermediário é chamada a partir do menu "*Agents*" e seleciona o endereço IP, porta e *community* do agente a ser consultado, assim como parâmetros do SNMP

para as consultas. A janela situada em primeiro plano mostra a monitoração da tabela de enlaces de um agente NBND sendo executado no endereço 127.0.0.14 na porta 2014. A tabela de enlaces apresentada representa uma topologia em anel com cinco nodos e quatro enlaces. Por exemplo, a primeira linha da tabela indica o contador 6, que significa enlace sem-falha, para o enlace entre os nodos 127.0.0.10 e 127.0.0.11.

O código completo do agente NBND para as duas estratégias de testes e os scripts auxiliares estão disponíveis nos anexos.

4.3 Visualização do Tráfego do Agente NBND na Rede

Esta seção descreve os procedimentos utilizados para analisar o tráfego na rede e visualizar os testes realizados pelos agentes para identificar eventos nos enlaces e as mensagens de disseminação emitidas pelos agentes quando da detecção de eventos. Os experimentos foram realizados sobre uma rede TCP/IP.

Para analisar as mensagens na rede foram utilizados dois softwares de *sniffer*, o *Tcpdump* e o *Ethereal*. Um sniffer é um aplicativo que executado em um host de uma rede de barramento compartilhado tem a capacidade de capturar todos os pacotes que trafegam neste barramento, e não somente os pacotes destinados ao host local conforme fazem os aplicativos comuns. O *Tcpdump* [25] é um *sniffer* de código aberto baseado na biblioteca *libpcap* com interface em modo texto. O *Ethereal* [24] possui uma poderosa interface gráfica e também utiliza a biblioteca *libpcap*. A *libpcap* [25] é uma biblioteca de captura de pacotes útil para monitoração de redes em baixo nível.

A figura 4.4 é um *screenshot* do aplicativo *Ethereal* utilizado para capturar pacotes durante a execução de diversos agentes NBND na mesma máquina local. Os agentes foram

configurados de forma que cada um utilize um endereço da interface *loopback* diferente, por exemplo, 127.0.0.10 ou 127.0.0.14. Na primeira das três janelas do aplicativo, são listados os pacotes com o instante de tempo em que foram capturados, endereços de origem e destino, além de outros campos do cabeçalho TCP/IP. O campo *time* identifica o tempo em que o pacote foi capturado da rede e é dado em segundos desde a captura do primeiro pacote pelo *Ethereal*.

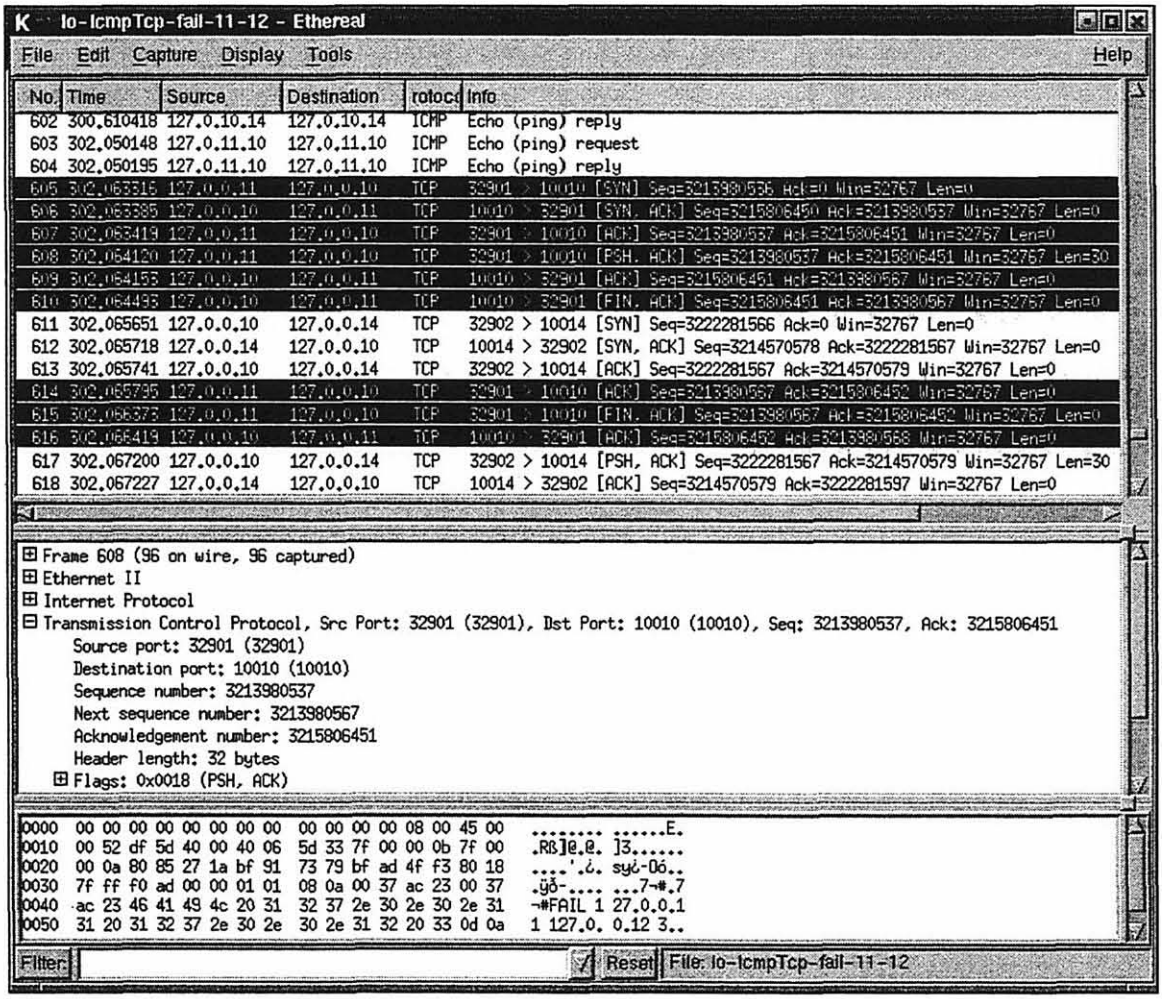


Figura 4.4: Captura das mensagens entre os agentes com o aplicativo *Ethereal*.

Na primeira janela da figura 4.4 observam-se três pacotes ICMP correspondentes aos testes entre os agentes e, em seguida, uma série de pacotes correspondentes à disseminação de

um evento de falha. Na segunda janela do aplicativo aparecem alguns dos campos do pacote 608. Na terceira janela, o pacote é mostrado em sua forma bruta em código hexadecimal e, mais à direita, em código ASCII.

Os pacotes de número 603 e 604 mostram um teste do nodo 127.0.0.11 para o nodo 127.0.0.10. Os 9 pacotes destacados do número 605 ao 616 representam a disseminação de uma mensagem do nodo 127.0.0.11 para o nodo 127.0.0.10 sobre o evento de falha no enlace entre os nodos 127.0.0.11 e 127.0.0.12. Conforme observado, para cada mensagem de disseminação 9 pacotes são colocados na rede. São 3 pacotes para a conexão de três vias do TCP/IP, 1 pacote do emissor para o receptor contendo a informação de disseminação, 1 pacote de *acknowledgement* do receptor para o emissor, e 4 pacotes para finalizar a conexão.

Para facilitar a visualização das mensagens de disseminação na rede, aplicamos um filtro para que em cada mensagem de disseminação apenas um dentre os 9 pacotes seja mostrado. O pacote que realmente contém as informações a respeito do evento é o quarto pacote na seqüência de 9 pacotes. No exemplo da figura 4.4 este pacote é o pacote de número 608. Para mostrar somente o pacote que contém as informações de cada disseminação, e mostrar as mensagens ICMP de teste de enlace, aplicam-se os seguintes filtros:

Filtro na sintaxe do Ethereal:

```
(tcp.flags.push == 1) || icmp
```

Filtro na sintaxe do Tcpdump:

```
tcpdump -qn -i lo -s 0 '(tcp[13] == 24) || icmp'
tcpdump -qn -r <arquivo.log> -s 0 '(tcp[13] == 24) || icmp'
```

O pacote de numero 608 tem seus campos expandidos na segunda janela do aplicativo *Ethereal*, e seu conteúdo é mostrado na terceira janela do aplicativo. Note-se que, dentre os 9 pacotes da mensagem, apenas o quarto pacote contém o *flag* "PUSH" acionado. Portanto o *flag* "PUSH" identifica um pacote único em cada mensagem de disseminação. Além disso, no conteúdo do quarto pacote, estão as informações a respeito do evento, quais sejam, os nodos

adjacentes ao enlace em que o evento ocorreu e o contador de eventos para o enlace. É incluída na mensagem de disseminação uma string, "FAIL" ou "REPAIR", para facilitar a identificação do evento, contudo esta string não faz parte da especificação do algoritmo NBND e poderia ser descartada sem prejuízo ao funcionamento do algoritmo.

No exemplo apresentado na figura 4.4, pode-se observar na terceira janela do aplicativo *Ethereal* a mensagem: "FAIL 127.0.0.11 127.0.0.12 3". Esta mensagem informa que o nodo 127.0.0.11 identificou um evento no enlace entre 127.0.0.11 e 127.0.0.12 e que o novo contador de eventos para este enlace é igual a 3. Como o contador de eventos é impar, sabe-se ainda, que o evento é uma falha.

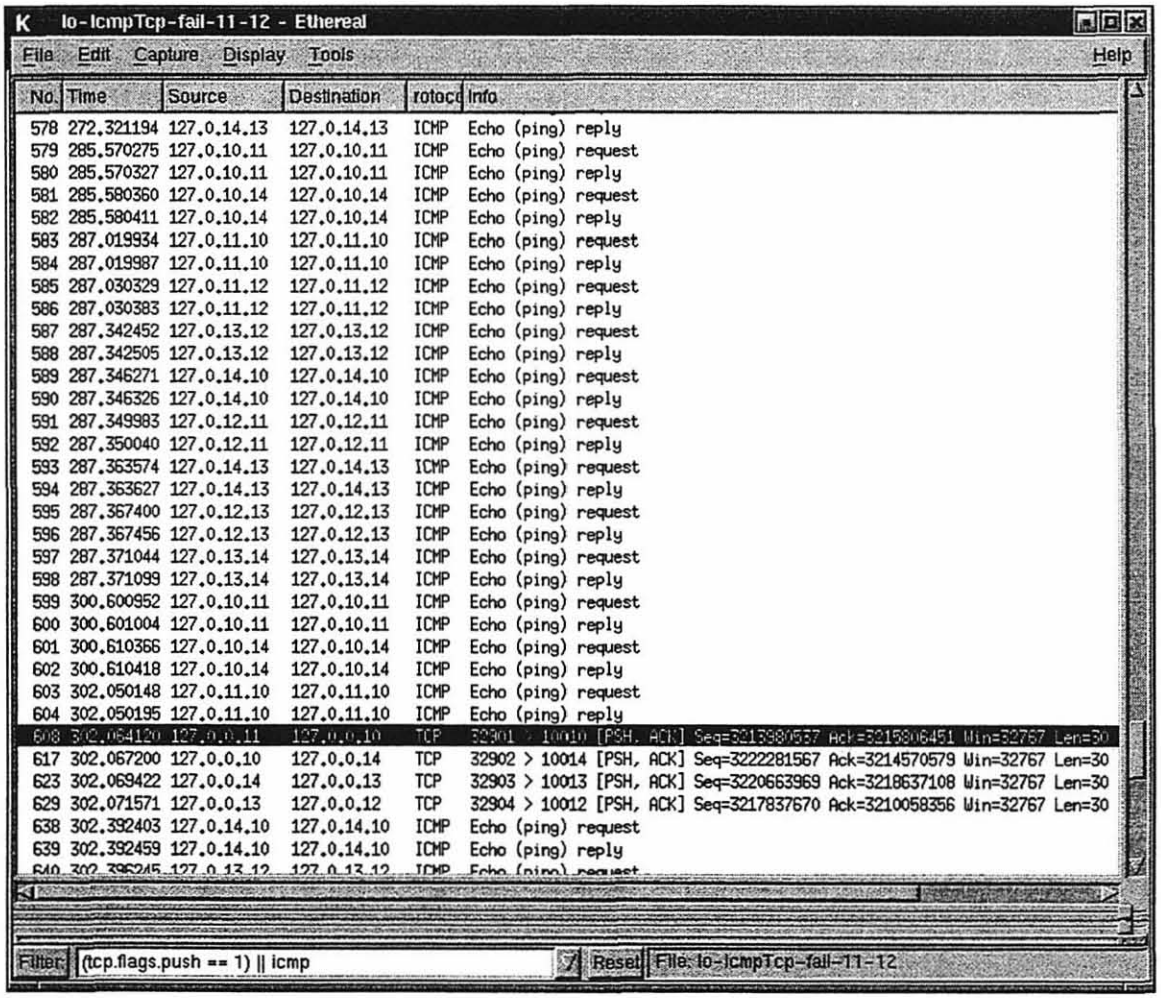


Figura 4.5: Mensagens de disseminação filtradas no Ethereal.

O mesmo fluxo de pacotes capturado pelo *Tcpdump* com a sintaxe "tcpdump -qn -i lo -s 0", ainda sem o filtro, é apresentado na figura 4.6. Com a aplicação do filtro, o *Tcpdump* produz a saída ilustrada na figura 4.7. A saída filtrada pelo *Ethereal* é mostrada na figura 4.5.

```
00:13:26.149438 127.0.10.14 > 127.0.10.14: icmp: echo reply
00:13:27.589168 127.0.11.10 > 127.0.11.10: icmp: echo request (DF)
00:13:27.589215 127.0.11.10 > 127.0.11.10: icmp: echo reply
00:13:27.602336 127.0.0.11.32901 > 127.0.0.10.10010: tcp 0 (DF)
00:13:27.602405 127.0.0.10.10010 > 127.0.0.11.32901: tcp 0 (DF)
00:13:27.602439 127.0.0.11.32901 > 127.0.0.10.10010: tcp 0 (DF)
00:13:27.603140 127.0.0.11.32901 > 127.0.0.10.10010: tcp 30 (DF)
00:13:27.603173 127.0.0.10.10010 > 127.0.0.11.32901: tcp 0 (DF)
00:13:27.603513 127.0.0.10.10010 > 127.0.0.11.32901: tcp 0 (DF)
00:13:27.604671 127.0.0.10.32902 > 127.0.0.14.10014: tcp 0 (DF)
00:13:27.604738 127.0.0.14.10014 > 127.0.0.10.32902: tcp 0 (DF)
00:13:27.604761 127.0.0.10.32902 > 127.0.0.14.10014: tcp 0 (DF)
00:13:27.604815 127.0.0.11.32901 > 127.0.0.10.10010: tcp 0 (DF)
00:13:27.605393 127.0.0.11.32901 > 127.0.0.10.10010: tcp 0 (DF)
00:13:27.605439 127.0.0.10.10010 > 127.0.0.11.32901: tcp 0 (DF)
00:13:27.606220 127.0.0.10.32902 > 127.0.0.14.10014: tcp 30 (DF)
00:13:27.606247 127.0.0.14.10014 > 127.0.0.10.32902: tcp 0 (DF)
```

Figura 4.6: Captura das mensagens entre os agentes com o aplicativo *Tcpdump*.

```
00:12:57.860214 127.0.14.13 > 127.0.14.13: icmp: echo reply
00:13:11.109295 127.0.10.11 > 127.0.10.11: icmp: echo request (DF)
00:13:11.109347 127.0.10.11 > 127.0.10.11: icmp: echo reply
00:13:11.119380 127.0.10.14 > 127.0.10.14: icmp: echo request (DF)
00:13:11.119431 127.0.10.14 > 127.0.10.14: icmp: echo reply
00:13:12.558954 127.0.11.10 > 127.0.11.10: icmp: echo request (DF)
00:13:12.559007 127.0.11.10 > 127.0.11.10: icmp: echo reply
00:13:12.569349 127.0.11.12 > 127.0.11.12: icmp: echo request (DF)
00:13:12.569403 127.0.11.12 > 127.0.11.12: icmp: echo reply
00:13:12.881472 127.0.13.12 > 127.0.13.12: icmp: echo request (DF)
00:13:12.881525 127.0.13.12 > 127.0.13.12: icmp: echo reply
00:13:12.885291 127.0.14.10 > 127.0.14.10: icmp: echo request (DF)
00:13:12.885346 127.0.14.10 > 127.0.14.10: icmp: echo reply
00:13:12.889003 127.0.12.11 > 127.0.12.11: icmp: echo request (DF)
00:13:12.889060 127.0.12.11 > 127.0.12.11: icmp: echo reply
00:13:12.902594 127.0.14.13 > 127.0.14.13: icmp: echo request (DF)
00:13:12.902647 127.0.14.13 > 127.0.14.13: icmp: echo reply
00:13:12.906420 127.0.12.13 > 127.0.12.13: icmp: echo request (DF)
00:13:12.906476 127.0.12.13 > 127.0.12.13: icmp: echo reply
00:13:12.910064 127.0.13.14 > 127.0.13.14: icmp: echo request (DF)
00:13:12.910119 127.0.13.14 > 127.0.13.14: icmp: echo reply
00:13:26.139972 127.0.10.11 > 127.0.10.11: icmp: echo request (DF)
00:13:26.140024 127.0.10.11 > 127.0.10.11: icmp: echo reply
00:13:26.149386 127.0.10.14 > 127.0.10.14: icmp: echo request (DF)
00:13:26.149438 127.0.10.14 > 127.0.10.14: icmp: echo reply
00:13:27.589168 127.0.11.10 > 127.0.11.10: icmp: echo request (DF)
00:13:27.589215 127.0.11.10 > 127.0.11.10: icmp: echo reply
00:13:27.603140 127.0.0.11.32901 > 127.0.0.10.10010: tcp 30 (DF)
00:13:27.606220 127.0.0.10.32902 > 127.0.0.14.10014: tcp 30 (DF)
00:13:27.608442 127.0.0.14.32903 > 127.0.0.13.10013: tcp 30 (DF)
00:13:27.610591 127.0.0.13.32904 > 127.0.0.12.10012: tcp 30 (DF)
00:13:27.931423 127.0.14.10 > 127.0.14.10: icmp: echo request (DF)
00:13:27.931479 127.0.14.10 > 127.0.14.10: icmp: echo reply
00:13:27.935265 127.0.13.12 > 127.0.13.12: icmp: echo request (DF)
```

Figura 4.7: Mensagens de disseminação filtradas no *Tcpdump*.

Para que o instante da ocorrência do evento possa ser detectado pelo *sniffer*, no momento do escalonamento do evento, é mandada uma mensagem ICMP *echo request* para um endereço especial.

4.4 Resultados Experimentais

Nesta seção, é mostrada a execução dos agentes NBND sobre uma topologia em anel com 5 nodos e sobre uma topologia de hipercubo de 8 nodos. Na topologia em anel, é ilustrada uma situação comparativa da detecção de eventos entre a estratégia de testes Two-way Test e a estratégia de testes Token Test. Na topologia do hipercubo de 8 nodos, é mostrada a detecção de uma falha com a estratégia Two-way Test e a sua disseminação através da árvore *breadth-first* a todos os nodos do sistema.

4.4.1 Estratégias de Detecção de Eventos

Na topologia em anel com 5 nodos e 4 enlaces, utilizando um intervalo de testes de 30 segundos, foi simulada a falha de um enlace na primeira metade do intervalo de testes. O instante da falha foi escolhido para demonstrar o menor tempo de detecção da estratégia de testes Token Test em comparação com a estratégia de testes Two-way Test, conforme discutido na seção 3.5. Devido à troca de mensagem que ocorre na metade do intervalo de testes no Token Test, a falha é detectada pelo Token Test na metade do intervalo de testes. Já no Two-way Test, a falha só é detectada ao final do intervalo de testes. A ocorrência da falha e a sua detecção pelo Two-way Test é ilustrada na figura 4.8. A detecção da falha pelo Token Test é mostrada na figura 4.9.

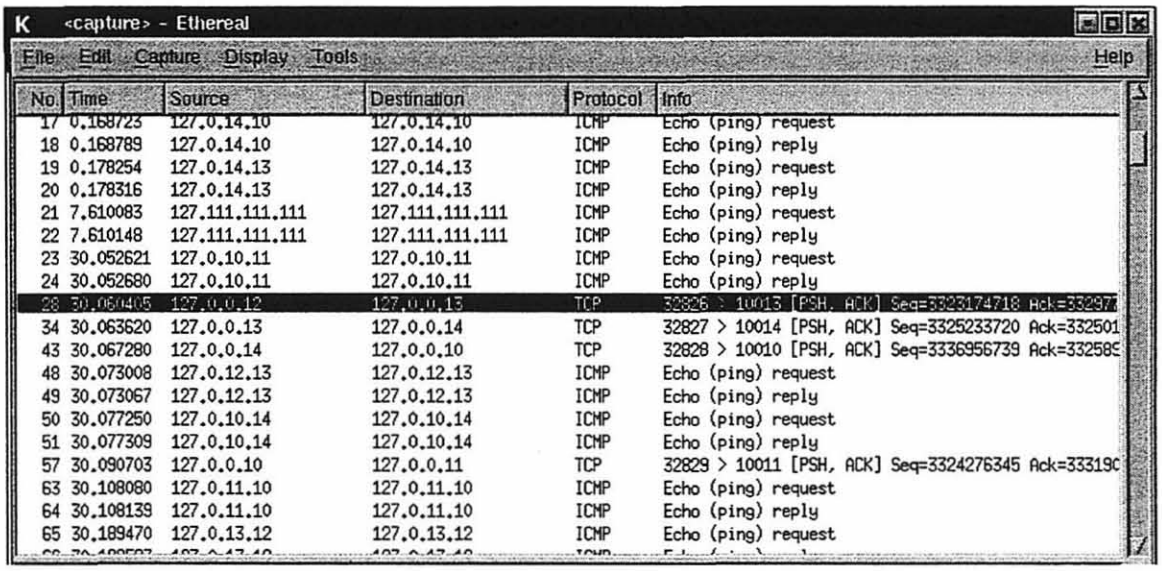


Figura 4.8: Detecção de falha de enlace com o Two-way Test.

Na figura 4.8, o momento da ocorrência da falha é indicado pela mensagem ICMP *echo request* para o endereço 127.111.111.111 no instante de tempo 7,610083 segundos, conforme o pacote de número 21. Logo que o evento é detectado pelo nodo 127.0.0.12, este manda uma mensagem de disseminação para o nodo 127.0.0.13, esta mensagem é representada pelo pacote de número 28, no instante de tempo 30,060405. Portanto o tempo de detecção da falha foi de aproximadamente 22,45 segundos.

No caso da estratégia Token Test, há a necessidade de uma forma de comunicação sem confirmação imediata, pois conforme descrito na seção 3.2, um nodo manda uma mensagem para um nodo adjacente, mas a resposta só é recebida depois de decorrido metade do intervalo de testes. Para satisfazer a esta propriedade, a estratégia de testes Token Test utiliza mensagens UDP, ao invés das mensagens ICMP *echo request* e ICMP *echo reply* usadas na estratégia Two-way Test.

Na figura 4.9, o momento da ocorrência da falha é indicado pela mensagem ICMP para o endereço 127.111.111.111 no instante de tempo 9,633397 segundos, indicada pelo pacote de número 6. O evento é detectado pelo nodo 127.0.0.12, o pacote em destaque é a primeira mensagem de disseminação, enviada ao nodo 127.0.0.13 no instante 18,021806 segundos. Portanto o tempo de detecção da falha foi de aproximadamente 8,39 segundos.

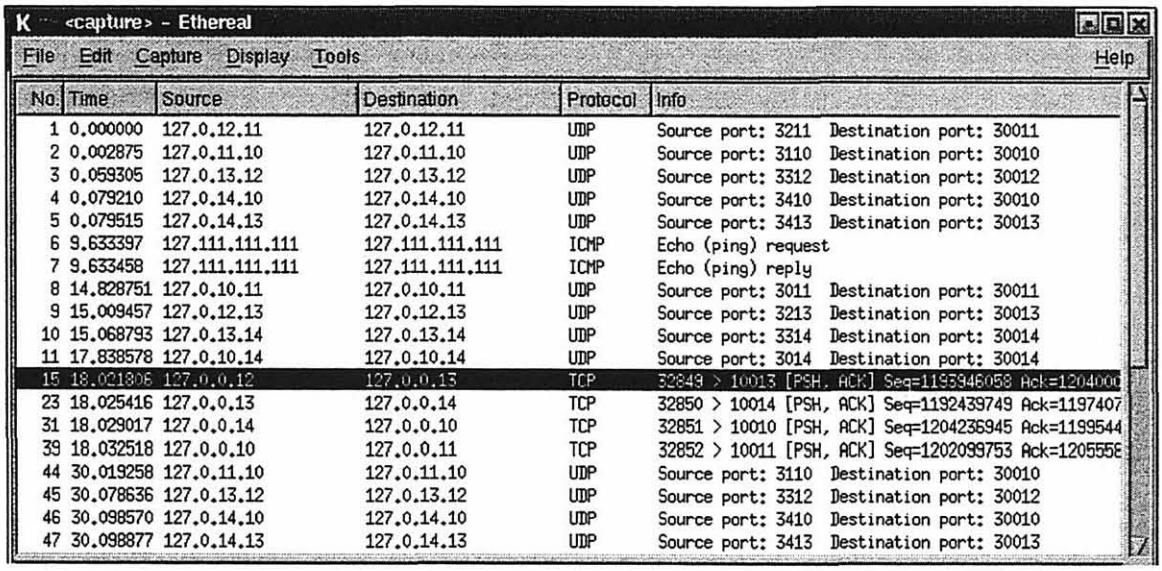


Figura 4.9: Detecção de falha de enlace com o Token Test.

Os tempos indicados pelo campo *time* do *Ethereal* na figura 4.8 e na figura 4.9 representam o tempo decorrido desde a captura do primeiro pacote. Em ambos os casos os agentes foram colocados em execução logo após o *Ethereal*, de modo que a primeira mensagem capturada, é a primeira mensagem da primeira rodada de testes dos agentes em execução. Desta forma o tempo indicado é muito próximo aos tempos de execução dos agentes. Como o intervalo de testes foi configurado nas duas estratégias para 30 segundos, no tempo 0, foi iniciado uma rodada de testes e no tempo 30 segundos iniciou-se a segunda rodada detestes. Assim o tempo 15 segundos marca a metade do intervalo de testes.

A detecção do evento deu-se no Token Test no instante 18, ou seja, 3 segundos após a metade do intervalo de testes. Estes 3 segundos são decorrentes de um tempo de tolerância, adicionado aos 15 segundos de meio intervalo de testes. Portanto o agente que não possui o token espera 18 segundos para checar se já recebeu o token, caso o token não tenha chegado o enlace é considerado falho. Este tempo de tolerância deve ser configurado para se adequar ao tempo de transmissão das mensagens na tecnologia de rede específica a fim de evitar alarmes falsos.

Comparando-se os tempos de detecção da estratégia Two-way Test e Token Test observa-se que o tempo de detecção do Two-way Test, que foi de 22,45 segundos, ficou entre meio intervalo de testes e um intervalo de testes. Já o tempo de detecção do Token Test, que foi de 8,39 segundos, ficou entre 0 e meio intervalo de testes, conforme descrito na situação *S1* da figura 3.7 da seção 3.5, que compara as duas estratégias de testes quando da ocorrência de um evento no instante de tempo entre 0 e meio intervalo de testes. Observe-se ainda que os tempos de detecção estão de acordo com a equação (1) da seção 3.5 que estabelece uma relação entre os tempos de detecção do Two-way Test e do Token Test para a situação *S1*. Ou seja, tempo de detecção do Token Test é igual ao tempo de detecção do Two-way Test somado de meio intervalo de testes.

4.4.1 Disseminação de Eventos

Para demonstrar a estratégia de disseminação de eventos do agente NBND são simuladas falhas nos enlaces na rede de topologia de hipercubo de 8 nodos. Uma falha é simulada de cada vez e são analisadas as mensagens de disseminação colocadas na rede pelos agentes NBND utilizando a estratégia de testes Two-way Test.

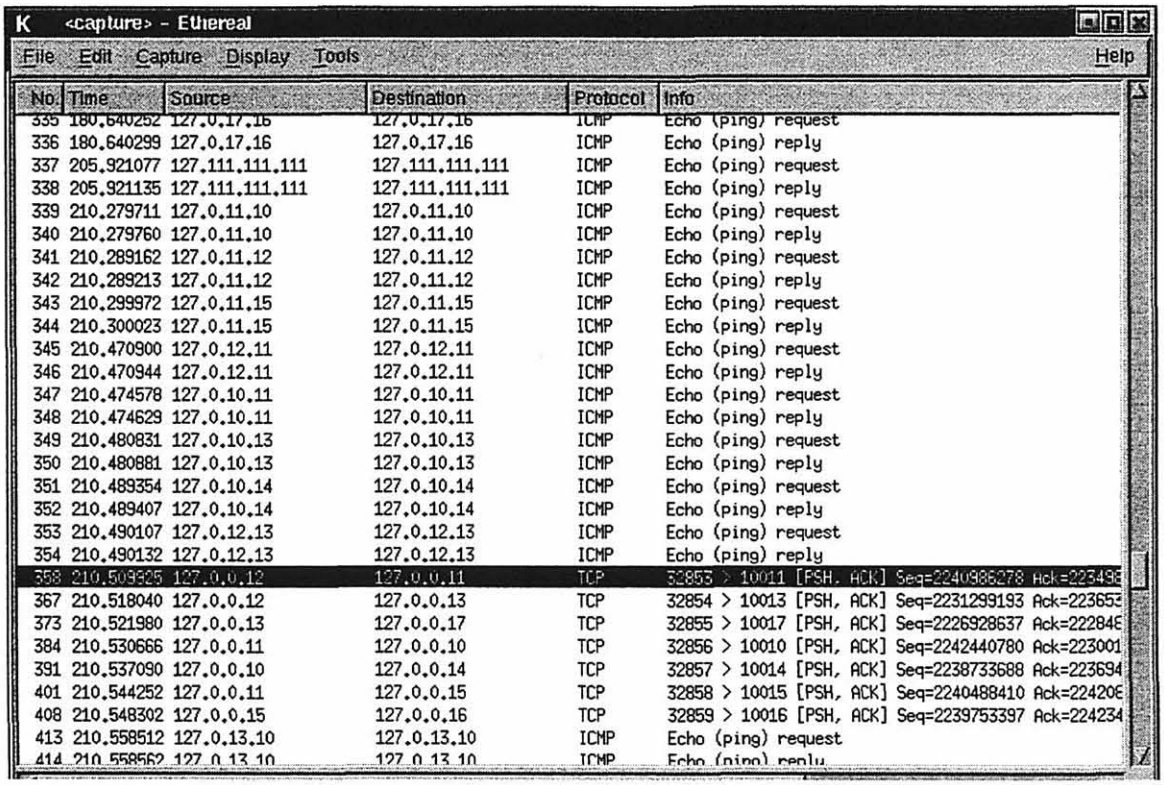


Figura 4.10: Disseminação de um evento na topologia do hipercubo de 8 nodos.

Na figura 4.10 pode-se observar o tráfego gerado na rede pela disseminação de uma falha ocorrida no enlace entre os nodos 127.0.0.12 e 127.0.0.16 na topologia do hipercubo de 8 nodos. A falha ocorre no instante de tempo 205,921077 segundos, conforme indicado pelo pacote ICMP de número 337. O evento é detectado pelo nodo 127.0.0.12 que monta uma árvore de disseminação *breadth-first* e dá inicio a disseminação mandando mensagens para seus dois filhos na árvore de disseminação. A primeira mensagem de disseminação é mandada para o nodo filho 127.0.0.11 no instante de tempo 210,509925 segundos, representada pelo pacote de número 358. A mensagem para o nodo 127.0.0.13 corresponde à mensagem de número 367.

A disseminação ocorre em paralelo para que todos os nodos do sistema fiquem sabendo do evento o mais rápido possível. O nodo 127.0.0.13 dissemina o evento para seu filho, o nodo 127.0.0.17, pelo pacote número 373. Em instante de tempo muito próximo, o nodo 127.0.0.11 manda mensagens para os seus filhos, os nodos 127.0.0.10 e 127.0.0.15. Estes por sua vez mandam a informação para seus filhos, o nodo 127.0.0.10 manda para o nodo 127.0.0.14 e o nodo 127.0.0.15 manda para o nodo 127.0.0.16, terminando a disseminação no instante 210,548302 segundos. É possível notar que a disseminação ocorre muito rapidamente se comparada com o intervalo de testes, são decorridos aproximadamente 0,038 segundos entre a primeira mensagem de disseminação e a última. Este tempo depende da profundidade da árvore de disseminação e do tempo de transmissão de cada mensagem. A profundidade depende da topologia da rede e o tempo de transmissão de cada mensagem depende da tecnologia da rede.

Capítulo 6

Conclusões e Perspectivas

Neste trabalho foi apresentada uma revisão detalhada dos algoritmos existentes para diagnóstico em nível de sistema. Uma nova versão do algoritmo NBND foi proposta e especificada formalmente. Foi apresentada uma comparação entre as duas estratégias de testes, a Two-way Test e a nova estratégia Token Test. Foi descrita a implementação de um sistema distribuído de monitoração de conectividade de redes de topologia arbitrária, com a utilização de agentes SNMP e da NBND MIB. Foram relatados casos de aplicação dos agentes NBND em duas topologias de rede: anel e hipercubo. O tráfego gerado pelas trocas de mensagens dos agentes foi observado, e com base neste foram analisadas as estratégias de detecção e de disseminação de eventos.

Como trabalhos futuros pretende-se criar uma interface gráfica para um cliente SNMP que faz o cálculo da conectividade com base nas informações consultadas na NBND MIB e mostra graficamente o mapa da rede e o estado de conectividade dos enlaces e nodos. Há a possibilidade de se implementar esta interface por meio da integração dos agentes NBND ao aplicativo TkIned que acompanha o pacote Scotty. O algoritmo necessita ainda de

mecanismos para tratar da ocorrência de eventos múltiplos. Por fim, pretende-se aplicar o algoritmo na monitoração de um *backbone* real, como o *backbone* da RNP.

Bibliografia

[1] BAGCHI, A.; e HAKIMI, S. L.. *An Optimal Algorithm for Distributed System-Level Diagnosis*, Proc. 21st Fault Tolerant Computing Symp., June, 1991.

[2] COMER, D. E.. *Interligação em Redes com TCP/IP – vol. 1*, tradução da 3ª ed., Campus, Rio de Janeiro, RJ, 1998.

[3] DUARTE JR, E. P.; CESTARI, J. M. A. P.. *O Agente Chinês para Diagnóstico de Redes de Topologia Arbitrária.* Anais do II workshop de Testes e Tolerância a Falhas da SBC, pp. 82-87, Curitiba, 2000.

[4] DUARTE JR, E. P.; MATTOS, G. O. *Diagnóstico de Redes de Topologia Arbitrária: Um algoritmo Baseado em Inundação de Mensagens.* Anais do II workshop de Testes e Tolerância a Falhas da SBC, pp. 82-87, Curitiba, 2000.

[5] DUARTE JR, E. P.. *Um Algoritmo para Diagnóstico de Redes de Topologia Arbitrária.* Anais do SBRC98, pp. 50-55, Porto Alegre, 1998.

[6] DUARTE JR, E. P.; MANSFIELD, G.; NANYA T. e NOGUCHI, S.. *Non-Broadcast Network Fault Monitoring Based on System-Level Diagnosis*, Proc. IEEE/IFIP IM'97, pp. 597-609, San Diego, May 1997.

[7] FABRIS, E.; SIQUEIRA, J. I.; DUARTE Jr, E. P.. *Simulação de um Algoritmo de Diagnóstico em Redes Ponto a Ponto*, 25° Congresso da Sociedade Brasileira de Computação, Anais do CTIC, pag 236, Rio de Janeiro, Julho de 1999.

[8] MASSON, G.; BLOUGH, D.; SULLIVAN G.. *System Diagnosis.* In Fault-Tolerant Computer System Design, ed. PRADHAN, D. K.. Prentice-Hall, 1996.

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

[10] ROSE, M. T.. *The Simple Book – An Introduction to Internet Management*, 2nd ed., PrenticeHall, Englewood Cliffs, NJ, 1994.

[11] SEDGEWICK, R.. *Algorithms in C*, Addison-Wesley, Reading, MA, 1992.

[12] SIQUEIRA, J. I.; FABRIS, E.; DUARTE JR, E. P.. *A Token-Based Testing Strategy for Non-Broadcast Network Diagnosis.* 1st IEEE Latin American Test Workshop - LATW2000, Rio de Janeiro - RJ, 2000.

[13] SIQUEIRA, J. I.; DUARTE JR, E. P.. *Uma Ferramenta para Diagnóstico de Redes de Alta Velocidade,*18º CSBRC - 2º WORKSHOP RNP2, 2000, Belo Horizonte - MG. 2000.

[14] STAHL, M.; BUSKENS, R. e BIANCHINI, R.. *Simulation of Adapt On-Lime Diagnosis Algorithm for General Topology Networks*, Proc. IEEE 11th Symp. Reliable Distributed Systems, October 1992.

[15] TANENBAUM, A. S.. *Redes de Computadores*, tradução da 3ª ed., Campus, Rio de Janeiro, RJ, 1997.

[16] TIVOLI SYSTEMS INC.. *Tivoli NetView.* http://www.tivoli.com/products/index/netview

[17] FURLAN, D.C. Comunicação Pessoal – Disseminação breadth-first modificada.

[18] PREPARATA, F.; METZE, G.; CHIEN, R.T., *On the connection assignment problem of diagnosable systems.* IEEE trans. Electr. Computer, EC-16 (1967) pp. 86-88.

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

[20] MAURO, D. R.; SCHMIDT, K. J.. *SNMP Essencial.* Ed. Campus, Rio de Janeiro, 2001.

[21] *Projeto Tcl.* http://sourceforge.net/projects/tcl

[22] *Projeto TkToolkit.* http://sourceforge.net/projects/tktoolkit

[23] *Biblioteca Scotty*. http://wwwsnmp.cs.utwente.nl/~schoenw/scotty

[24] *Ethereal*. http://www.ethereal.com

[25] *Aplicativo Tcpdump / Biblioteca libpcap*. http://www.tcpdump.org

[26] *Conectiva Linux*. http://www.conectiva.com.br

[27] *Firewall Iptables*. http://www.iptables.org

[28] MULLENDER, S.. *Distributed Systems*.Addison-Wesley, New York, 1994.

[29] LEINWAND, A.; FANG-CONROY, K.. *Network Management: A Practical Perspective*. Addison-Wesley, Reading, MA, 1995.

Apêndices

Este apêndice traz os programas utilizados para a implementação do sistema de monitoramento baseado no algoritmo NBND. São mostrados ainda alguns scripts utilizados para controlar a execução dos agentes e simular as falhas nos enlaces da rede.

Scripts para Controlar a Execução dos Agentes

```
##############################################################################
# usage.txt --->
# Este Arquivo documenta como iniciar os agentes nbnd da topologia de rede
# configurada na maquina local.
# Sao apresentadas algumas formas de controlar a execucao dos agentes
# e simular eventos
##############################################################################

- Inicia os agentes na topologia previamente configurada:
  % ./nbnd-launch

- Mata os agentes:
  % ./nbnd-stop

- Lista os processos:
  % ./nbnd-list

- Visualiza a mib usando o mibtree:
  % ./nbnd-view

- Para "derrubar" um enlace entre o nodo 127.0.0.<no1> e 127.0.0.<no2>
  (Onde 1 < no1,no2 < 99); (A ordem dos nos nao importa):
  % nbnd-linkDown <no1> <no2>

- Para restaurar o enlace
  % nbnd-linkUp <no1> <no2>

- Para listar as regras no firewall que simulam as falhas nos enlaces
  % nbnd-filterList

- Para disparar os agentes nos nodos de uma topologia <topo> certifique-se
  de que:
     1- nbnd-launch aponta para nbnd-launch-<topo>
     2- nbnd-conf.tcl aponta para nbnd-conf-<topo>.tcl
  Exemplo:
  nbnd-lauch apontando para nbnd-launch-anel5
  Dispara 5 agentes na maquina local usando a interface loopback
  nos enderecos 127.0.0.10 a 127.0.0.14 nas respetivas portas 2010 a 2014.
  (No exemplo abaixo, a saida eh direcionada para dbg-anel5.log)
  % ./nbnd-launch 2>dbg.log

- Para alterar a topologia:
  % <edit> nbnd-conf-<topologia>.tcl
  % ln -sf nbnd-conf-<topologia>.tcl nbnd-conf.tcl

- Para visualizar o log do agente "IP-n":
  % cat dbg-<topo>.log | grep ^<IP-n> | less

- Para startar um unico agente num endereco <IP-n> na porta <Port-y>:
  % ./nbnd.tcl <IP-n> <Port-y>

- Para percorrer toda a mib com o snmpwalk (NET-SNMP)
  % snmpwalk -p <Port-y> <IP-n> public .1

- Para remediar problema do "unable to alloc" (tcl-8.3.3 / scotty-2.1.11)
  % rm -f /usr/lib/tnm2.1.11/i686-Linux-2.4.5-9cl/*
```

nbnd.launch-anel5

```
#!/bin/sh
# Starts the agents in specified IP's and ports:
# ./nbnd.tcl <IP> <PORT>
./nbnd.tcl 127.0.0.10 2010 &
./nbnd.tcl 127.0.0.11 2011 &
./nbnd.tcl 127.0.0.12 2012 &
./nbnd.tcl 127.0.0.13 2013 &
./nbnd.tcl 127.0.0.14 2014 &
```

nbnd.launch-cubo3

```
#!/bin/sh
# Starts the agents in specified IP's and ports:
# ./nbnd.tcl <IP> <PORT>
./nbnd.tcl 127.0.0.10 2010 &
./nbnd.tcl 127.0.0.11 2011 &
./nbnd.tcl 127.0.0.12 2012 &
./nbnd.tcl 127.0.0.13 2013 &
./nbnd.tcl 127.0.0.14 2014 &
./nbnd.tcl 127.0.0.15 2015 &
./nbnd.tcl 127.0.0.16 2016 &
./nbnd.tcl 127.0.0.17 2017 &
```

nbnd.list

```
ps axw | grep nbnd.tcl
```

nbnd.stop

```
kill `ps auxw | grep "scotty ./nbnd" | awk '{print $2}'`
```

nbnd-conf-anel5.tcl

```
# Anel de 5 nodos, 5 arestas
####################################
# Configuracao inicial da topologia
####################################
global no1
global no2
global IP

#########################
# Mapeamento nodos->IPs :
#########################
set IP(0) 127.0.0.10
set IP(1) 127.0.0.11
set IP(2) 127.0.0.12
set IP(3) 127.0.0.13
set IP(4) 127.0.0.14

#########################
# Portas dos agentes:
```

```
#########################
set Porta(0) 2010
set Porta(1) 2011
set Porta(2) 2012
set Porta(3) 2013
set Porta(4) 2014

set portaIP(127.0.0.10) 2010
set portaIP(127.0.0.11) 2011
set portaIP(127.0.0.12) 2012
set portaIP(127.0.0.13) 2013
set portaIP(127.0.0.14) 2014

#########################
# Lista de arestas:
#########################
set no1(0) 0
set no2(0) 1

set no1(1) 0
set no2(1) 4

set no1(2) 1
set no2(2) 0

set no1(3) 1
set no2(3) 2

set no1(4) 2
set no2(4) 1

set no1(5) 2
set no2(5) 3

set no1(6) 3
set no2(6) 2

set no1(7) 3
set no2(7) 4

set no1(8) 4
set no2(8) 0

set no1(9) 4
set no2(9) 3
```

nbnd-conf-cubo3.tcl

```
# Hiper cubo de grau 3: 8 nodos; 12 arestas
###################################
# Configuracao inicial da topologia
###################################
global no1
global no2
global IP

#########################
# Mapeamento nodos->IPs :
#########################
set IP(0) 127.0.0.10
set IP(1) 127.0.0.11
set IP(2) 127.0.0.12
set IP(3) 127.0.0.13
set IP(4) 127.0.0.14
set IP(5) 127.0.0.15
set IP(6) 127.0.0.16
set IP(7) 127.0.0.17

#########################
# Portas dos agentes:
```

```
########################
set Porta(0) 2010
set Porta(1) 2011
set Porta(2) 2012
set Porta(3) 2013
set Porta(4) 2014
set Porta(5) 2015
set Porta(6) 2016
set Porta(7) 2017

set portaIP(127.0.0.10) 2010
set portaIP(127.0.0.11) 2011
set portaIP(127.0.0.12) 2012
set portaIP(127.0.0.13) 2013
set portaIP(127.0.0.14) 2014
set portaIP(127.0.0.15) 2015
set portaIP(127.0.0.16) 2016
set portaIP(127.0.0.17) 2017

########################
# Lista de arestas:
########################
set no1(0) 0
set no2(0) 1

set no1(1) 0
set no2(1) 3

set no1(2) 0
set no2(2) 4

set no1(3) 1
set no2(3) 0

set no1(4) 1
set no2(4) 2

set no1(5) 1
set no2(5) 5

set no1(6) 2
set no2(6) 1

set no1(7) 2
set no2(7) 3

set no1(8) 2
set no2(8) 6

set no1(9) 3
set no2(9) 0

set no1(10) 3
set no2(10) 2

set no1(11) 3
set no2(11) 7

set no1(12) 4
set no2(12) 0

set no1(13) 4
set no2(13) 5

set no1(14) 4
set no2(14) 7

set no1(15) 5
set no2(15) 1

set no1(16) 5
set no2(16) 4

set no1(17) 5
```

```
set no2(17) 6

set no1(18) 6
set no2(18) 2

set no1(19) 6
set no2(19) 5

set no1(20) 6
set no2(20) 7

set no1(21) 7
set no2(21) 3

set no1(22) 7
set no2(22) 4

set no1(23) 7
set no2(23) 6
```

Scripts para Simular Falhas nos Enlaces da Rede

nbnd.filterDown

```
#!/bin/sh
# Cria filtros no firewall simulando uma falha no link entre <node1> e <node2>
# USAGE: ./nbnd.filterDown <node1> <node2>
# ---
# O ping de um nodoX para um nodoY eh montado como um ping para
# 127.0.<nodoX>.<nodoY>
#  --> Esta foi a forma encontrada para poder simular, no firewall, falhas
#      num link rodando todos os nodos na maquina local
# O ping para 127.111.111.111 eh para possibilitar a identificacao do momento
# da falha atraves da analise do trafego da rede

# Usado no TwoWay-Test com ping
#fping 127.111.111.111
#iptables -A OUTPUT -p icmp -d 127.0.$1.$2 -j DROP
#iptables -A OUTPUT -p icmp -d 127.0.$2.$1 -j DROP

# Usado no Token-Test com UDP
#fping 127.111.111.111
#iptables -A OUTPUT -p udp -d 127.0.$1.$2 -j DROP
#iptables -A OUTPUT -p udp -d 127.0.$2.$1 -j DROP

# Resolve pros dois:
fping 127.111.111.111
iptables -A OUTPUT -d 127.0.$1.$2 -j DROP
iptables -A OUTPUT -d 127.0.$2.$1 -j DROP
```

nbnd.filterUp

```
#!/bin/sh
# Remove filtros no firewall da falha no link entre <node1> e <node2>
# USAGE: ./nbnd.filterUp <node1> <node2>
# ---
# O ping de um nodoX para um nodoY eh montado como um ping para
```

```
# 127.0.<nodoX>.<nodoY>
#  --> Esta foi a forma encontrada para poder simular, no firewall, falhas
#      num link rodando todos os nodos na maquina local
# O ping para 127.222.222.222 eh para possibilitar a identificacao do momento
# da recuperacao atraves da analise do trafego da rede

# Usado no TwoWay-Test com ping
#fping 127.111.222.222
#iptables -D OUTPUT -p icmp -d 127.0.$1.$2 -j DROP
#iptables -D OUTPUT -p icmp -d 127.0.$2.$1 -j DROP

# Usado no Token-Test com UDP
#fping 127.111.222.222
#iptables -D OUTPUT -p udp -d 127.0.$1.$2 -j DROP
#iptables -D OUTPUT -p udp -d 127.0.$2.$1 -j DROP

# Resolve pros dois:
fping 127.111.222.222
iptables -D OUTPUT -d 127.0.$1.$2 -j DROP
iptables -D OUTPUT -d 127.0.$2.$1 -j DROP
```

nbnd.filterList

```
# lista as regras do firewall onde sao simuladas as quedas de links
iptables -L
```

Script para Analizar a Saída do Tcpdump

```
#!/usr/bin/tclsh
# Usage:
# % ./nbnd-parse.tcl <arquivo de log do sniffer>
# Escreve em "<arq de log>.tDetect"
##########################################################################
# Le^  o arquivo gerado pelo tcpdump das mensagens de teste e disseminacao
# de eventos. Identifica o momento do escalonamento da falha pela msgm
# especial (ping 127.111.111.111 = fail) ou (ping 127.111.222.222 = repair)
# Calcula o tempo entre a ocorrencia da falha e a deteccao desta por algum
# nodo. O tempo de deteccao da falha eh jogado no arquivo de saida
# uma deteccao por linha.
##########################################################################

append fileOut1 [lindex $argv 0] ".tDetect"
puts stdout "-> $fileOut1..."
set out1 [open $fileOut1 w]
set in [open [lindex $argv 0] r]
set disseminando 0

while { [gets $in new] >= 0 } {
    #puts [lindex $new 0]
    ## Se jah ocorreu o evento:
    if { $disseminando == 1 } {
          ## Procura 1o pacote de disseminacao
          if { [string compare [lindex $new 4] "tcp"] == 0 } {
              ## Calcula tempo de deteccao
              set fimStamp [lindex $new 0]
              set tDiss [expr $fimStamp - $iniStamp]
              puts $out1 "$tDiss"
              set disseminando 0
          }
    ## Se evento ainda nao ocorreu, espera evento:
    } else {
          ## Se endereco de origem eh <x>.111.<x>.<x> eh um novo evento:
```

```
            if {[lindex [split [lindex $new 1] .] 1] == 111 } {
                set iniStamp [lindex $new 0]
                #puts $iniStamp
                set disseminando 1
            }
        }
}
close $out1
close $in

#  proc time2Mili { timeIn } {
#      set inteiro [lindex [split $timeIn .] 0]
#      set mili [lindex [split $timeIn .] 1]
#      set hora [lindex [split $inteiro :] 0]
#      set min [lindex [split $inteiro :] 1]
#      set sec [lindex [split $inteiro :] 2]
#          set ret [expr $mili + ($sec*1000000) + ($min *60000000) +
($hora*3600000000)]
#      return $ret
#  }
```

Código da Estratégia Two-way Test

```
#!/bin/sh
# nbnd.tcl
# the next line restarts using scotty -*- tcl -*- \
exec scotty "$0" "$@"
package require Tnm 2.1

####################
# Vars Globais:
####################
global IP             # vetor contendo os IPs dos nodos IP(0)=127.0.0.10
global no1          # vetor contendo os primeiros nodos das arestas
global no2          # no2(0) contem o segundo nodo da aresta 0
global nodeIP1        # vetor contendo os primeiros IPs das arestas
global nodeIP2        # nodeIP2(0) contem o segundo IP da aresta 0
global eventCounter # vetor com contador para os enlaces (indexado por int)
global loop         # contador de loops

global listIP
global Lneigh        # Vetor de listas ordenadas dos vizinhos de cda nodo
global myNeigh       # Lista fixa com os vizinhos do nodo local (uso em test)
####################

proc CreateAgent { version ip port {interp {}} } {
    global tnm
    #set ip [netdb hosts address $tnm(host)]
    set s [snmp session -port $port -address $ip]
    if {[catch {$s configure -version $version} msg]} {
        $s destroy
        error $msg
    }
    if {$version == "SNMPv2U"} {
        $s configure -user $tnm(user)
    }
    $s configure -agent $interp
    return $s
}


#############################################################
# Funçao que identifica filhos na Arvore breadth-first
#############################################################
# Retorna lista de filhos na arvore breadth-first
#    visitados(no) --> controla qdo todos os nos jah foram visitados ou:
```

```
#    LFilhos --> fila contendo os filhos do no atual
#############################################################
proc bftFilhos { noRaiz } {
    global listIP
    global visitados
    global myIP
    # Zera lista de retorno
    set Lret {}
    # Zera lista de filhos/irmaos LFilhos
    set LFilhos {}
    # zera vetor visitados
    foreach noS $listIP {
          set visitados($noS) 0
    }
    # comeca pela raiz
    set noS $noRaiz

    # Enquanto nao atingir todos os nos:
    while { $noS != -1 } {
          visitNo $noS
          lappend LFilhos $noS
          # Enquanto houver nos filhos
          while { [llength $LFilhos] != 0 } {
              set noX [retira LFilhos]
              set noND [firstSucc $noX]
              #puts stderr "$myIP :DBG: RETORNO de firstSucc: $noND"
              # Enquanto houver irmaos deste no para serem visitados:
              while {$noND != -1} {
             if { $visitados($noND) == 0 } {
                 visitNo $noND
                 #puts stderr "$myIP :DBG: noX: $noX -  noND:$noND"
                 lappend LFilhos $noND
                 if { $noX == $myIP } {
               lappend Lret $noND
                 }
             }
             set noND [nextSucc $noX $noND]
             #puts stderr "$myIP :DBG: RETORNO de nextSucc: $noND"
               }
          }
          # Soh eh necessario qdo for desconexo
          set noS [selectNo]
    }
    return $Lret
}


# Visita $no
#########################
proc visitNo { noX } {
    global myIP
    global visitados
    set visitados($noX) 1
    #puts stderr "$myIP :DBG: nodo visitado: $noX"
}

# Retira o primeiro elemento da fila
####################################
proc retira { lista } {
    global myIP
    upvar $lista lista2
    set elemento [lindex $lista2 0]
    set lista2 [lrange $lista2 1 end]
    #puts stderr "$myIP :RETIRA: lista resultante = $lista2"
    return $elemento
}

# Deleta elemento "elem2" da lista de vizinhos Lneigh(elem1), e
# Deleta elemento "elem1" da lista de vizinhos Lneigh(elem2)
# Mantem as listas ordenadas
#####################################################################
proc delNeigh { elem1 elem2 } {
    global myIP
    global Lneigh
```

```
    set listAux {}
    #puts stderr "$myIP :DELNEIGH in: elem1: $elem1 ; elem2: $elem2"
    #puts stderr "$myIP :DELNEIGH in: Lneigh($elem1): $Lneigh($elem1)"
    foreach node $Lneigh($elem1) {
        if { [string compare $node $elem2] != 0 } {
            lappend listAux $node
        }
    }
    set Lneigh($elem1) $listAux
    set listAux {}
    foreach node $Lneigh($elem2) {
        if { [string compare $node $elem1] != 0 } {
            lappend listAux $node
        }
    }
    set Lneigh($elem2) $listAux
    #puts stderr "$myIP :DELNEIGH out: Lneigh($elem1): $Lneigh($elem1)"
    #puts stderr "$myIP :DELNEIGH out: Lneigh($elem2): $Lneigh($elem2)"
}

# Insere elemento na lista de adjacencias, ordenando-a
#############################################################
proc insNeigh { lista elemento } {
    upvar $lista lista2
    lappend lista2 $elemento
    set lista2 [lsort $lista2]
}

# Retorna o primeiro vizinho do noX na lista de adjacencias Lneigh
####################################################################
proc firstSucc { noX } {
    global Lneigh
    global myIP
    if { [llength $Lneigh($noX)] != 0 } {
        set ret [lindex $Lneigh($noX) 0]
    } else {
        set ret -1
    }
    #puts stderr "$myIP :FIRSTSUCC ( $noX ) = $ret"
    return $ret
}

# Retorna o vizinho do noX seguinte ao noY em Lneigh
#####################################################
proc nextSucc { noX noY } {
    global Lneigh
    global myIP
    set ret -1
    set proximo 0
    #puts stderr "$myIP :NEXTSUCC in: Lneigh($noX) = $Lneigh($noX)"
    foreach noZ $Lneigh($noX) {
        if { $proximo == 1} {
            set ret $noZ
            set proximo 0
        }
        if { $noZ == $noY } {
            set proximo 1
        }
    }
    #puts stderr "$myIP :NEXTSUCC ( $noX $noY ) = $ret"
    return $ret
}

# Retorna o proximo no ainda nao visitado
#    Esta funcao eh usada na breadth-first caso o grafo seja desconexo
#########################################################################
proc selectNo { } {
    global listIP
    global visitados
    set ret -1
    foreach noZ $listIP {
        if { $visitados($noZ) == 0 } {
            set ret $noZ
        }
```

```
    }
    return $ret
}

#############################################################################
### Testa os nodos vizinhos
###
### - Dispara teste para todos os vizinhos
### - Antes de testar identifica se o link para o vizinho estava falho
### - Os testes sao assincronos, assim espera todos terminarem em vwait
#############################################################################
proc test {} {
    global myNeigh
    global myIP
    global eventCounter
    global i_counterIP
    #global pingging
    #global fim_ping
    global loop
    incr loop
    puts stderr "$myIP : Testing loop: $loop..."
    #set fim_ping 0
    #set pingging [llength $myNeigh]
    foreach nodeIP $myNeigh {
          set estavaFalho 0
          set myCount $eventCounter([set i_counterIP($myIP,$nodeIP)])
          if { [expr $myCount % 2] == 1} {
              set estavaFalho 1
          }
          icmpNoBlk_test $nodeIP $estavaFalho $myCount
    }
    # espera todos os testes para os vizinhos terminarem
    #vwait fim_ping
}

# Dispara um ping nao bloqueante para o endereco 127.0.<myIP>.<nodoDst>
#  - Este esquema de endereco eh usado para poder bloquear no firewall local
#############################################################################
proc icmpNoBlk_test { nodeIP estavaFalho myCount } {
    global myIP
    global fp
    set ping "/usr/sbin/fping"
    set addr "127.0.[lindex [split $myIP .] 3].[lindex [split $nodeIP .] 3]"
    set command "|$ping $addr "
    if {[catch {open $command} channel]} {
        puts "'$command' must not be right for you."
    }
    set fp($nodeIP) $channel
    fconfigure $channel -buffering none
    fileevent $channel readable [handle_icmp $nodeIP $estavaFalho $myCount]
}

# Quando canal do ping retorna, trata retorno do ping
# - Se link "down" ou "up" for um novo evento, dissemina evento
#############################################################
proc handle_icmp { nodeIP estavaFalho myCount } {
    global myIP
    #global pingging
    global fp
    set channel $fp($nodeIP)
    ## Trata erro no canal
    if { [eof $channel] || [catch {gets $channel lineIn} result1] } {
        puts stderr "$myIP : HANDLE_ICMP vai fechar canal!!"
          if { [catch {close $channel} result2] } {
              puts stderr "$myIP : HANDLE_ICMP erro ao FECHAR canal!!"
          }
    ## Se nao houve erro no canal do ping:
    } else {
        #close $channel
          if { [string compare [lindex $lineIn 2] "alive"] } {
              if { $estavaFalho == 0 } {
             ## Se Nao estava falho e ping falhou --> diss. falha
             puts stderr "$myIP :: Link( $myIP:$nodeIP ) DISS DOWN!!!"
             set new [expr $myCount + 1]
```

```
            disseminaSock "FAIL" $myIP $nodeIP $new
              }
          } else {
              if { $estavaFalho == 1 } {
            ## Se estava falho e ping deu ok --> diss. recuperacao
            puts stderr "$myIP :: Link( $myIP:$nodeIP ) DISS RECOVER!!!."
            set new [expr $myCount + 1]
            disseminaSock "REPAIR" $myIP $nodeIP $new
              }
          }
    }
    ## controla qdo todos os pings para os vizinhos ja' retornaram
    #incr pingging -1
    #if { [string compare $pingging "0"] == 0 } {
    #     set fim_ping 1
    #}
}


## Configura Socket servidor para recebimento de msgms de disseminacao
proc serverDiss { channel chost cport } {
    global myIP
    puts stderr "$myIP :: <--- MSGM from $chost"
    fconfigure $channel -buffering none
    fileevent $channel readable [list recDiss $channel]
}

## Trata msgm de disseminacao recebida
proc recDiss { channel } {
    global myIP
    global errorCode errorInfo
    #puts stderr "$myIP :: vai Receber MSGM"
    if { [eof $channel] || [catch {gets $channel msgIn}]} {
          puts stderr "$myIP : RECDISS fechando canal!!"
          close $channel
          return
    }
    puts stderr "$myIP :: MSGM recebida: $msgIn"
    ## Espera processar a msgm anterior ou um evento recem detectado
    ## talvez tenha q trocar isto por um semaforo bem delimitado com vwait
    #update
    close $channel
    set err [catch {
          # Se for msgm de disseminacao de Falha:
          if { [string compare [lindex $msgIn 0] "FAIL"] == 0 } {
              set eventoIn [lindex $msgIn 0]
              set noRaizIn [lindex $msgIn 1]
              set noFalhoIn [lindex $msgIn 2]
              set counterIn [lindex $msgIn 3]

              disseminaSock $eventoIn $noRaizIn $noFalhoIn $counterIn
          }
          if { [string compare [lindex $msgIn 0] "REPAIR"] == 0 } {
              set eventoIn [lindex $msgIn 0]
              set noRaizIn [lindex $msgIn 1]
              set noFalhoIn [lindex $msgIn 2]
              set counterIn [lindex $msgIn 3]
              #antes de disseminar o repair tem q atualizar o nodo
              disseminaSock $eventoIn $noRaizIn $noFalhoIn $counterIn
          }
    }]
    if {$err != 0} {
          puts stderr "$myIP : ERRO em recDiss!!"
          puts stderr "$myIP : ...errorCode=$errorCode, errorInfo=$errorInfo"
    }
}

#######################################################
# DISSEMINACAO (Socket)
#
# Se o novo count for mais novo que o atual dissemina.
#   Usado tanto na disseminaçao inical pelo nodo q detecta
#   qto pelos nodos no meio da arvore
```

```
# Para cada filhos na arvore)
#   Manda msgm contendo [nodo raiz : nodo falho : counter]
#########################################################
proc disseminaSock { evento noRaiz noFalho newCount } {
    global Lneigh
    global myIP
    global i_counterIP
    global eventCounter
    #Identifica o contador local correspondente
    set myCount $eventCounter([set i_counterIP($noRaiz,$noFalho)])
    if { $newCount > $myCount } {
          #puts stderr "$myIP :: newCount novo: Need Update"
          #obtem indice dos counters referentes ao link em questao
          set count1 $eventCounter([set i_counterIP($noRaiz,$noFalho)])
          set count2 $eventCounter([set i_counterIP($noFalho,$noRaiz)])
          #puts stderr "$myIP :DBG: count1 (Raiz,Falho): $count1"
          #puts stderr "$myIP :DBG: count2 (Falho,Raiz): $count2"
          ## Atualiza counters locais e Lista Adj. (Lneigh):
          set eventCounter([set i_counterIP($noRaiz,$noFalho)]) $newCount
          set eventCounter([set i_counterIP($noFalho,$noRaiz)]) $newCount
          if { [string compare $evento "FAIL"] == 0 } {
              delNeigh $noRaiz $noFalho
          } else {
              insNeigh Lneigh($noRaiz) $noFalho
              insNeigh Lneigh($noFalho) $noRaiz
          }
          ## Dissemina para filhos na arvore Breadth First
          #puts stderr "$myIP :DBG: bftFilhos em noRaiz: $noRaiz"
          set filhos [bftFilhos $noRaiz]
          #puts stderr "$myIP :DBG: DISSEMINAR para: $filhos"
          if { [llength $filhos] > 0 } {
              foreach IPDest $filhos {
             puts stderr "$myIP : Dissemina para $IPDest --->"
             mandaDiss $IPDest $evento $noRaiz $noFalho $newCount
              }
          } else {
              if { [string compare $noRaiz $myIP] == 0 } {
             puts stderr "$myIP : Nodo INATINGIVEL!! fazer ACK"
              } else {
             puts stderr "$myIP : No FOLHA atingido!! fazer ACK"
              }
          }
    } else {
          puts stderr "$myIP : newCount VELHO: Dont Update : OLDINFO"
          ## Nao continua Dissem
    }
}

proc mandaDiss {IPDest evento noRaiz noFalho newCount} {
global portaIP myIP
global errorCode errorInfo
    set err [catch {
          set portDest [expr 8000 + $portaIP($IPDest)]
          set chDissOut [socket -myaddr $myIP $IPDest $portDest]
          puts $chDissOut "$evento $noRaiz $noFalho $newCount"
          flush $chDissOut
          close $chDissOut
    }]
    if { $err != 0} {
          puts stderr "$myIP : ERRO em mandaDiss!!"
          puts stderr "$myIP : ...errorCode=$errorCode, errorInfo=$errorInfo"
    }
    return $err
}

#################################
## Corpo Principal do Programa
## - Inicializacoes diversas
## - Dispara os testes
#################################
#################################

## - Configura saida padrao de erros
fconfigure stderr -buffering none
```

```
## - Load my confs from stdin
#############################
if { $argc == 2 } {
    set myIP [lindex $argv 0]
    set myPort [lindex $argv 1]
    puts stderr "Meu IP: $myIP, porta: $myPort"
} else {
    puts stderr "Usage: $argv0 <Local_IP> <Local_Port>"
    exit
}

## interprete NAO seguro
set interp ""

## cria o agente usando um interprete NAO seguro !
set Agent [CreateAgent SNMPv1 $myIP $myPort $interp]
## carrega MIB de diagnostico NBND
mib load ./diagnosis.mib

#####################################
## Inicializa MIB
## - source nbnd_conf, wich fulfills topology with neighbors
## - Stores topology in the MIB
#####################################
$Agent instance diagTableDescr.0 diagTableDescr "NBND table - This table keeps the
states of nodes been monitored by the NBND Agents"

# Le topologia
source ./nbnd-conf.tcl

# Armazena topologia na MIB
foreach i [array names no1] {
    $Agent instance node1.$i nodeIP1($i) $IP($no1($i))
    $Agent instance node2.$i nodeIP2($i) $IP($no2($i))
    $Agent instance eventCounter.$i eventCounter($i) 0
    # seta vetores auxiliares para ident. de indice do counter
    # set ind_no1($nodeIP1($i)) $i
    # set ind_no2($nodeIP2($i)) $i
    set i_counterIP($nodeIP1($i),$nodeIP2($i)) $i
}
#####################################################################
# Configura porta para receber disseminacao e trata a disseminacao
#####################################################################
## Cria um socket na porta (8000 + myPort) para receber msgs de disseminacao
set dissPort [expr 8000 + $myPort]
#puts stderr "$myIP :: Abrindo socket na porta: $dissPort"
set err [catch {socket -server serverDiss $dissPort} msg]
if {$err != 0} {
    global myIP
    puts stderr "$myIP :: Erro abrindo socket na porta $dissPort: $msg"
}


#####################################
# 1- Cria lista de adjacencias, Lneigh(<IP>) contendo todos os vizinhos
# de todos os nodos (OBS: Qdo aresta DOWN ela eh removido desta lista).
# 2- Cria lista dos meus vizinhos (myNeigh) para ser usada no momento
# dos testes.
# (OBS: Esta lista eh fixa, mesmo qdo aresta DOWN)
#####################################
# Cria lista de IPs ordenada
foreach i [array names IP] {
    lappend listIPtmp $IP($i)
}
set listIP [lsort $listIPtmp]
#puts stderr "$myIP :DBG: Lista de IPs ordenada: $listIP"

# Cria lista de adjacencias:
foreach noIP $listIP {
    foreach ind [array names nodeIP1] {
        if { $nodeIP1($ind) == $noIP } {
            lappend neighTmp($noIP) $nodeIP2($ind)
        }
```

```
    }
}
# Ordena lista:
foreach noIP $listIP {
    set Lneigh($noIP) [lsort $neighTmp($noIP)]
}
set myNeigh $Lneigh($myIP)
puts stderr "$myIP :: Lista dos meus vizinhos: $myNeigh"
#puts stderr "$myIP :DBG: Lista de TODOS os vizinhos: $Lneigh"


#Imprime Topologia da rede pela lista de adjacencias:
puts stderr "$myIP :: ----------------------"
puts stderr "$myIP :: - TOPOLOGIA DA REDE: -"
puts stderr "$myIP :: ----------------------"
foreach noIP $listIP {
    puts stderr "$myIP : Vizinhos do no: $noIP:"
    foreach adj $Lneigh($noIP) {
          puts stderr "$myIP :: $adj"
    }
    puts stderr "$myIP :: ----------------------"
}

# ---
set loop 0

sleep 3

while {1} {
    test
    set proceed false
    after 30000 {set proceed true}
    vwait proceed
}
```

Código da Estratégia Token Test

```
#!/bin/sh
# nbnd.tcl
# the next line restarts using scotty -*- tcl -*- \
exec scotty "$0" "$@"
package require Tnm 2.1

##############################################################################
# Vars Globais:
##############################################################################
global IP             # vetor contendo os IPs dos nodos IP(0)=127.0.0.10
global no1          # vetor contendo os primeiros nodos das arestas
global no2          # no2(0) contem o segundo nodo da aresta 0
global nodeIP1        # vetor contendo os primeiros IPs das arestas
global nodeIP2        # nodeIP2(0) contem o segundo IP da aresta 0
global eventCounter # vetor com contador para os enlaces (indexado por int)
global loop         # contador de loops
global NeighLoop    # contador de loops por vizinho

global listIP
global Lneigh       # Vetor de listas ordenadas dos vizinhos de cda nodo
global myNeigh      # Lista fixa com os vizinhos do nodo local (uso em test)
##############################################################################
global fdTknSnd     # Vetor de fds udp para mandar token
global baseTknSend  # Porta base para mandar token
global portTknSend     # Vetor de portas UDP locais usadas para mandar token
                       # cada porta = (base + nodeID*100 + nodeIDdestino)
global fdTknRec     # File descriptor do socket udp de recebimento de tokens
global baseTknRec   # Porta base para receber token
global portTknRec      # Vetor de portas UDP usadas para receber o token
```

```
                        # Cada nodo tem uma porta (base + nodeID)
#############################################################################

#####################################################
# Portas base usadas para mandar e receber tokens
#####################################################
set baseTknSend 2000
set baseTknRec 30000

#####################################################
# Parametros para o intervalo de testes do TokenTest
#####################################################
set interval 15
set extra 3

## Funcao que ccria um agente escutando na porta especificada
proc CreateAgent { version ip port {interp {}} } {
    global tnm
    #set ip [netdb hosts address $tnm(host)]
    set s [snmp session -port $port -address $ip]
    if {[catch {$s configure -version $version} msg]} {
        $s destroy
        error $msg
    }
    if {$version == "SNMPv2U"} {
        $s configure -user $tnm(user)
    }
    $s configure -agent $interp
    return $s
}


#############################################################
# Funçao que identifica filhos na Arvore breadth-first
#############################################################
# Retorna lista de filhos na arvore breadth-first
#    visitados(no) --> controla qdo todos os nos jah foram visitados ou:
#    LFilhos --> fila contendo os filhos do no atual
#############################################################
proc bftFilhos { noRaiz } {
    global listIP
    global visitados
    global myIP
    # Zera lista de retorno
    set Lret {}
    # Zera lista de filhos/irmaos LFilhos
    set LFilhos {}
    # zera vetor visitados
    foreach noS $listIP {
          set visitados($noS) 0
    }
    # comeca pela raiz
    set noS $noRaiz

    # Enquanto nao atingir todos os nos:
    while { $noS != -1 } {
          visitNo $noS
          lappend LFilhos $noS
          # Enquanto houver nos filhos
          while { [llength $LFilhos] != 0 } {
              set noX [retira LFilhos]
              set noND [firstSucc $noX]
              #puts stderr "$myIP :DBG: RETORNO de firstSucc: $noND"
              # Enquanto houver irmaos deste no para serem visitados:
              while {$noND != -1} {
             if { $visitados($noND) == 0 } {
                 visitNo $noND
                 #puts stderr "$myIP :DBG: noX: $noX -  noND:$noND"
                 lappend LFilhos $noND
                 if { $noX == $myIP } {
               lappend Lret $noND
                 }
             }
             set noND [nextSucc $noX $noND]
```

```
            #puts stderr "$myIP :DBG: RETORNO de nextSucc: $noND"
              }
          }
          # Soh eh necessario qdo for desconexo
          set noS [selectNo]
    }
    return $Lret
}

#########################
# Visita $no
proc visitNo { noX } {
    global myIP
    global visitados
    set visitados($noX) 1
    #puts stderr "$myIP :DBG: nodo visitado: $noX"
}

# Retira o primeiro elemento da fila
####################################
proc retira { lista } {
    global myIP
    upvar $lista lista2
    set elemento [lindex $lista2 0]
    set lista2 [lrange $lista2 1 end]
    #puts stderr "$myIP :RETIRA: lista resultante = $lista2"
    return $elemento
}

# Deleta elemento "elem2" da lista de vizinhos Lneigh(elem1), e
# Deleta elemento "elem1" da lista de vizinhos Lneigh(elem2)
# Mantem as listas ordenadas
#####################################################################
proc delNeigh { elem1 elem2 } {
    global myIP
    global Lneigh
    set listAux {}
    #puts stderr "$myIP :DELNEIGH in: elem1: $elem1 ; elem2: $elem2"
    #puts stderr "$myIP :DELNEIGH in: Lneigh($elem1): $Lneigh($elem1)"
    foreach node $Lneigh($elem1) {
          if { [string compare $node $elem2] != 0 } {
              lappend listAux $node
          }
    }
    set Lneigh($elem1) $listAux
    set listAux {}
    foreach node $Lneigh($elem2) {
          if { [string compare $node $elem1] != 0 } {
              lappend listAux $node
          }
    }
    set Lneigh($elem2) $listAux
    #puts stderr "$myIP :DELNEIGH out: Lneigh($elem1): $Lneigh($elem1)"
    #puts stderr "$myIP :DELNEIGH out: Lneigh($elem2): $Lneigh($elem2)"
}

# Insere elemento na lista de adjacencias, ordenando-a
######################################################################
proc insNeigh { lista elemento } {
    upvar $lista lista2
    lappend lista2 $elemento
    set lista2 [lsort $lista2]
}

# Retorna o primeiro vizinho do noX na lista de adjacencias Lneigh
#####################################################################
proc firstSucc { noX } {
    global Lneigh
    global myIP
    if { [llength $Lneigh($noX)] != 0 } {
          set ret [lindex $Lneigh($noX) 0]
    } else {
          set ret -1
    }
```

```
    #puts stderr "$myIP :FIRSTSUCC ( $noX ) = $ret"
    return $ret
}

# Retorna o vizinho do noX seguinte ao noY em Lneigh
#####################################################
proc nextSucc { noX noY } {
    global Lneigh
    global myIP
    set ret -1
    set proximo 0
    #puts stderr "$myIP :NEXTSUCC in: Lneigh($noX) = $Lneigh($noX)"
    foreach noZ $Lneigh($noX) {
        if { $proximo == 1} {
            set ret $noZ
            set proximo 0
        }
        if { $noZ == $noY } {
            set proximo 1
        }
    }
    #puts stderr "$myIP :NEXTSUCC ( $noX $noY ) = $ret"
    return $ret
}

# Retorna o proximo no ainda nao visitado
#    Esta funcao eh usada na breadth-first caso o grafo seja desconexo
#####################################################################
proc selectNo { } {
    global listIP
    global visitados
    set ret -1
    foreach noZ $listIP {
        if { $visitados($noZ) == 0 } {
            set ret $noZ
        }
    }
    return $ret
}

#####################################################################
### Testa os nodos vizinhos
###
### - Dispara teste para todos os vizinhos
### - Antes de testar identifica se o link para o vizinho estava falho
### - Os testes sao assincronos, assim espera todos terminarem em vwait
#####################################################################
proc test {} {
    global myNeigh
    global myIP
    global eventCounter
    global i_counterIP
    #global pingging
    #global fim_ping
    global loop
    incr loop
    puts stderr "$myIP : Testing loop: $loop..."
    #set fim_ping 0
    #set pingging [llength $myNeigh]
    foreach nodeIP $myNeigh {
        set estavaFalho 0
        set myCount $eventCounter([set i_counterIP($myIP,$nodeIP)])
        if { [expr $myCount % 2] == 1} {
            set estavaFalho 1
        }
        icmpNoBlk_test $nodeIP $estavaFalho $myCount
    }
    # espera todos os testes para os vizinhos terminarem
    #vwait fim_ping
}

# Dispara um ping nao bloqueante para o endereco 127.0.<myIP>.<nodoDst>
#  - Este esquema de endereco eh usado para poder bloquear no firewall local
#####################################################################
```

```
proc icmpNoBlk_test { nodeIP estavaFalho myCount } {
    global myIP
    global fp
    set ping "/usr/sbin/fping"
    set addr "127.0.[lindex [split $myIP .] 3].[lindex [split $nodeIP .] 3]"
    set command "|$ping $addr "
    if {[catch {open $command} channel]} {
        puts "myIP : ERRO em '$command'"
    }
    set fp($nodeIP) $channel
    fconfigure $channel -buffering none
    fileevent $channel readable [handle_icmp $nodeIP $estavaFalho $myCount]
}

# Quando canal do ping retorna, trata retorno do ping
# - Se link "down" ou "up" for um novo evento, dissemina evento
#################################################################
proc handle_icmp { nodeIP estavaFalho myCount } {
    global myIP
    #global pingging
    global fp
    set channel $fp($nodeIP)
    ## Trata erro no canal
    if { [eof $channel] || [catch {gets $channel lineIn} result1] } {
        puts stderr "$myIP : HANDLE_ICMP vai fechar canal!!"
           if { [catch {close $channel} result2] } {
               puts stderr "$myIP : HANDLE_ICMP erro ao FECHAR canal!!"
           }
    ## Se nao houve erro no canal do ping:
    } else {
        #close $channel
           if { [string compare [lindex $lineIn 2] "alive"] } {
               if { $estavaFalho == 0 } {
              ## Se Nao estava falho e ping falhou --> diss. falha
              puts stderr "$myIP :: Link( $myIP:$nodeIP ) DISS DOWN."
              set new [expr $myCount + 1]
              disseminaSock "FAIL" $myIP $nodeIP $new
               }
           } else {
               if { $estavaFalho == 1 } {
              ## Se estava falho e ping deu ok --> diss. recuperacao
              puts stderr "$myIP :: Link( $myIP:$nodeIP ) DISS RECOVER."
              set new [expr $myCount + 1]
              disseminaSock "REPAIR" $myIP $nodeIP $new
               }
           }
    }
    ## controla qdo todos os pings para os vizinhos ja' retornaram
    #incr pingging -1
    #if { [string compare $pingging "0"] == 0 } {
    #     set fim_ping 1
    #}
}


#####################################################################
# Teste com Token para UM vizinho
#
#    Se tem o token --> Manda Token; escalona novo teste
#    Senao --> checa se recebeu o Token
#        Escalona evento
#        Se nao recebeu e eh novo evento --> dissemina FAIL
#        Se recebeu e eh novo evento     --> dissemina REPAIR
#
#####################################################################
proc tokenTest { nodeIP } {
    global interval
    global extra
    global Token
    global tokenIn
    global myIP
    global eventCounter
    global i_counterIP
    global NeighLoop
```

```
    incr NeighLoop($nodeIP)
    puts stderr "$myIP :: Testing $nodeIP loop: $NeighLoop($nodeIP)..."
    set estavaFalho 0
    set myCount $eventCounter([set i_counterIP($myIP,$nodeIP)])
    if { [expr $myCount % 2] == 1} {
          set estavaFalho 1
    }
    ## Se esta com o token --> manda token
    if { $Token($nodeIP) == 1 } {
          sendToken $nodeIP
          set tokenIn($nodeIP) 0
          set Token($nodeIP) 0
          after [expr $interval * 1000] tokenTest $nodeIP
          ##after [expr $interval * 1000] { tokenTest $nodeIP }
    ## Se o vizinho esta com o token --> checa o recebimento
    } else {
          after [expr $interval * 1000] tokenTest $nodeIP
          ##after [expr $interval * 1000] { tokenTest $nodeIP }
          ## Existe um tempo de tolerancia para chegada do token
          #puts stderr "$myIP :: pausa $extra SECS para recebimento de $nodeIP"
          set proceed false
          after [expr $extra * 1000] {set proceed true}
          vwait proceed
          set Token($nodeIP) 1
          #sleep 3
          ## Se token nao chegou:
          if { $tokenIn($nodeIP) == 0 } {
              puts stderr "$myIP :: <-X- TOKEN de $nodeIP nao chegou !!!!"
              if { $estavaFalho == 0 } {
             ## Se nao estava falho e o tokenTest falhou --> diss. falha:
             puts stderr "$myIP :: Link( $myIP:$nodeIP ) DISS DOWN."
             set new [expr $myCount + 1]
             disseminaSock "FAIL" $myIP $nodeIP $new
              }
          } else {
              puts stderr "$myIP :: <- TOKEN de $nodeIP chegou"
              if { $estavaFalho == 1 } {
             ## Se estava falho e token foi recebido --> diss. recuperacao:
             puts stderr "$myIP :: Link( $myIP:$nodeIP ) DISS RECOVER."
             set new [expr $myCount + 1]
             disseminaSock "REPAIR" $myIP $nodeIP $new
              }
          }
    }
}

# Funcao que manda um TOKEN (pacote UDP) para o vizinho
#   Manda para 127.0.<noSrc>.<noDst>
#   Usa porta local = base + noSrc*100 + noDst
#   Usa porta remota = base + noDst
######################################################################
proc sendToken { nodeIP } {
    global portTknRec
    global fdTknSnd
    global myIP
    puts stderr "$myIP :: Mandando TOKEN de $myIP para $nodeIP -> "
    ## usa o endereco 127.0.<noSRC>.<noDST>
    set addr "127.0.[lindex [split $myIP .] 3].[lindex [split $nodeIP .] 3]"
    ## usa a porta local em "$fdTknSnd" para mandar para "$portTknRec($nodeIP)"
    catch {
          udp send $fdTknSnd($nodeIP) $addr $portTknRec($nodeIP) "TOKEN de $myIP"
    }
}

# Funcao que recebe um TOKEN (pacote UDP) de um vizinho e indica este
# recebimento em um buffer que sera depois checado
# --> Ou faz uma funcao para todos os vizinhos na mesma porta
# --> ou utiliza uma porta pra cada vizinho e carrega a funcao uma vez
#     para cada vizinho
######################################################################
proc recebeToken { } {
    global myIP
    global fdTknRec
    global tokenIn
```

```
    set msgTokenIn [lindex [udp receive $fdTknRec] 2]
    #puts stderr "$myIP :: Msgm Recebida: '$msgTokenIn'"
    if {[string compare [lindex $msgTokenIn 0] "TOKEN"] == 0} {
          set nodeRem [lindex $msgTokenIn 2]
          set tokenIn($nodeRem) 1

    }
}

##############################################################################
# - Inicia o vetor de tokens: Os nodos comecam testando os menores q ele
#     Token(<IP1>) indica se o nodo local possui o token para testar o IP1
#     se Token(<IP1>) == 0 entao e' o IP1 que realiza o teste
# - Inicia os contadores de loop por vizinhos
# - Inicia as portas de saida para cada vizinho
#     Porta para cada vizinho = (baseTknSend + (myId*100) + NodeId)
##############################################################################
proc initToken {} {
    global tokenIn
    global Token
    global NeighLoop
    global myNeigh
    global myIP
    global fdTknSnd
    global portTknSend
    global portTknRec
    global baseTknSend
    global baseTknRec
    global myPort
    foreach neigh $myNeigh {
          ## Se meu identificador maior que o do vizinho --> eu comeco o teste
          if { [string compare $myIP $neigh] == 1 } {
              set Token($neigh) 1
          } else {
              set Token($neigh) 0
          }
          ## Zera todos os buffers de recebimento de tokens
          set tokenIn($neigh) 0
          ## Inicia os contadores de loop para todos os vizinhos
          set NeighLoop($neigh) 0

          ## Seta as portas de envio para todos os vizinhos
          set nodeId [lindex [split $neigh .] 3]
          set myId [lindex [split $myIP .] 3]
          set portTknSend($neigh) [expr $baseTknSend + ($myId * 100) + $nodeId]

          ## Inicia os fds dos sockets UDP de saida para todos os vizinhos
          set fdTknSnd($neigh) [udp open $portTknSend($neigh)]

          ## Seta as portas de recebimento de todos os vizinhos
          set portTknRec($neigh) [expr $baseTknRec + $nodeId]
    }
}

# Dispara a funcao recebeToken quando chegar pacote UDP na
# porta "portTokenRec" vindo de algum vizinho
#  - Porta de recebimento = baseTknRec + myID
##############################################################################
proc initRecebeToken { } {
    global myIP
    global portTknRec
    global baseTknRec
    global fdTknRec
    set myId [lindex [split $myIP .] 3]
    set portTknRec($myIP) [expr $baseTknRec + $myId]
    set fdTknRec [udp open $portTknRec($myIP)]
    #puts stderr "$myIP :: Socket de recebeToken na porta: $portTknRec($myIP)"
    udp bind $fdTknRec readable { recebeToken }
}

## Configura Socket servidor para recebimento de msgms de disseminacao
proc serverDiss { channel chost cport } {
    global myIP
    puts stderr "$myIP :: <--- MSGM from $chost:$cport"
```

```
    fconfigure $channel -buffering none
    fileevent $channel readable [list recDiss $channel]
}

## Trata msgm de disseminacao recebida
proc recDiss { channel } {
    global myIP
    global errorCode errorInfo
    #puts stderr "$myIP :: vai Receber MSGM"
    if { [eof $channel] || [catch {gets $channel msgIn}]} {
        puts stderr "$myIP : RECDISS fechando canal!!"
        close $channel
        return
    }
    puts stderr "$myIP :: MSGM recebida: $msgIn"
    ## Espera processar a msgm anterior ou um evento recem detectado
    ## talvez tenha q trocar isto por um semaforo bem delimitado com vwait
    #update
    close $channel
    set err [catch {
        # Se for msgm de disseminacao de Falha:
        if { [string compare [lindex $msgIn 0] "FAIL"] == 0 } {
            set eventoIn [lindex $msgIn 0]
            set noRaizIn [lindex $msgIn 1]
            set noFalhoIn [lindex $msgIn 2]
            set counterIn [lindex $msgIn 3]
            disseminaSock $eventoIn $noRaizIn $noFalhoIn $counterIn
        }
        if { [string compare [lindex $msgIn 0] "REPAIR"] == 0 } {
            set eventoIn [lindex $msgIn 0]
            set noRaizIn [lindex $msgIn 1]
            set noFalhoIn [lindex $msgIn 2]
            set counterIn [lindex $msgIn 3]
            #antes de disseminar o repair tem q atualizar o nodo
            disseminaSock $eventoIn $noRaizIn $noFalhoIn $counterIn
        }
    }]
    if {$err != 0} {
        puts stderr "$myIP : ERRO em recDiss!!"
        puts stderr "$myIP : ...errorCode=$errorCode, errorInfo=$errorInfo"
    }
}

#########################################################
# DISSEMINACAO (Socket)
#
# Se o novo count for mais novo que o atual dissemina.
#   Usado tanto na disseminaçao inical pelo nodo q detecta
#   qto pelos nodos no meio da arvore
# Para cada filhos na arvore)
#   Manda msgm contendo [nodo raiz : nodo falho : counter]
#########################################################
proc disseminaSock { evento noRaiz noFalho newCount } {
    global Lneigh
    global myIP
    global i_counterIP
    global eventCounter
    #Identifica o contador local correspondente
    set myCount $eventCounter([set i_counterIP($noRaiz,$noFalho)])
    if { $newCount > $myCount } {
        #puts stderr "$myIP :DBG: newCount novo: Need Update"
        #obtem indice dos counters referentes ao link em questao
        set count1 $eventCounter([set i_counterIP($noRaiz,$noFalho)])
        set count2 $eventCounter([set i_counterIP($noFalho,$noRaiz)])
        ## Atualiza counters locais e Lista Adj. (Lneigh):
        set eventCounter([set i_counterIP($noRaiz,$noFalho)]) $newCount
        set eventCounter([set i_counterIP($noFalho,$noRaiz)]) $newCount
        if { [string compare $evento "FAIL"] == 0 } {
            delNeigh $noRaiz $noFalho
        } else {
            insNeigh Lneigh($noRaiz) $noFalho
            insNeigh Lneigh($noFalho) $noRaiz
        }
        ## Dissemina para filhos na arvore Breadth First
```

```
            #puts stderr "$myIP :DBG: bftFilhos em noRaiz: $noRaiz"
            set filhos [bftFilhos $noRaiz]
            #puts stderr "$myIP :DBG: DISSEMINAR para: $filhos"
            if { [llength $filhos] > 0 } {
                foreach IPDest $filhos {
               puts stderr "$myIP : Dissemina para $IPDest --->"
               mandaDiss $IPDest $evento $noRaiz $noFalho $newCount
                }
            } else {
                if { [string compare $noRaiz $myIP] == 0 } {
               puts stderr "$myIP : Nodo INATINGIVEL!! fazer ACK"
                } else {
               puts stderr "$myIP : No FOLHA atingido!! fazer ACK"
                }
            }
    } else {
            puts stderr "$myIP :: newCount VELHO: Dont Update: OLDINFO"
            ## Nao continua Dissem
    }
}

proc mandaDiss {IPDest evento noRaiz noFalho newCount} {
global portaIP myIP
global errorCode errorInfo
    set err [catch {
            set portDest [expr 8000 + $portaIP($IPDest)]
            set chDissOut [socket -myaddr $myIP $IPDest $portDest]
            puts $chDissOut "$evento $noRaiz $noFalho $newCount"
            flush $chDissOut
            close $chDissOut
    }]
    if { $err != 0} {
            puts stderr "$myIP : ERRO em mandaDiss!!"
            puts stderr "$myIP : ...errorCode=$errorCode, errorInfo=$errorInfo"
    }
    return $err
}

##################################
# Corpo Principal do Programa
##################################

##################################
## Inicializacões diversas:
##################################
# Configura saida padrao de erros
fconfigure stderr -buffering none

### - Load my confs from stdin
##############################
if { $argc == 2 } {
    set myIP [lindex $argv 0]
    set myPort [lindex $argv 1]
    puts stderr "Meu IP: $myIP, porta: $myPort"
} else {
    puts stderr "Usage: $argv0 <Local_IP> <Local_Port>"
    exit
}

## interprete NAO seguro
set interp ""

## cria o agente usando um interprete NAO seguro !
set Agent [CreateAgent SNMPv1 $myIP $myPort $interp]
## carrega MIB de diagnostico NBND
mib load ./diagnosis.mib

### Loads initial confs
###
### - source nbnd_conf, wich fulfills topology with neighbors
### - Stores topology in the MIB

#####################################
# Inicializa MIB
```

```
#######################################
$Agent instance diagTableDescr.0 diagTableDescr "NBND table - This table keeps the
states of nodes been monitored by the Distributed Network Conectivity tool "

# Le topologia
source ./nbnd-conf.tcl

# Armazena topologia na MIB
foreach i [array names no1] {
    $Agent instance node1.$i nodeIP1($i) $IP($no1($i))
    $Agent instance node2.$i nodeIP2($i) $IP($no2($i))
    $Agent instance eventCounter.$i eventCounter($i) 0
    # seta vetores auxiliares para ident. de indice do counter
    # set ind_no1($nodeIP1($i)) $i
    # set ind_no2($nodeIP2($i)) $i
    set i_counterIP($nodeIP1($i),$nodeIP2($i)) $i
}

#######################################
# 1- Cria lista de adjacencias, Lneigh(<IP>) contendo todos os vizinhos
# de todos os nodos (OBS: Qdo aresta DOWN ela eh removido desta lista).
# 2- Cria lista dos meus vizinhos (myNeigh) para ser usada no momento
# dos testes.
# (OBS: Esta lista eh fixa, mesmo qdo aresta DOWN)
#######################################
# Cria lista de IPs ordenada
foreach i [array names IP] {
    lappend listIPtmp $IP($i)
}
set listIP [lsort $listIPtmp]
#puts stderr "$myIP :DBG: Lista de IPs ordenada: $listIP"

# Cria lista de adjacencias:
foreach noIP $listIP {
    foreach ind [array names nodeIP1] {
        if { $nodeIP1($ind) == $noIP } {
            lappend neighTmp($noIP) $nodeIP2($ind)
        }
    }
}
# Ordena lista:
foreach noIP $listIP {
    set Lneigh($noIP) [lsort $neighTmp($noIP)]
}
set myNeigh $Lneigh($myIP)
puts stderr "$myIP :: Lista dos meus vizinhos: $myNeigh"
#puts stderr "$myIP :DBG: Lista de TODOS os vizinhos: $Lneigh"


#Imprime Topologia da rede pela lista de adjacencias:
puts stderr "$myIP :: ----------------------"
puts stderr "$myIP :: - TOPOLOGIA DA REDE: -"
puts stderr "$myIP :: ----------------------"
foreach noIP $listIP {
    puts stderr "$myIP : Vizinhos do no: $noIP:"
    foreach adj $Lneigh($noIP) {
        puts stderr "$myIP :: $adj"
    }
    puts stderr "$myIP :: ----------------------"
}

#####################################################################
# Configura porta para receber disseminacao e trata a disseminacao
#####################################################################

## Cria um socket na porta (8000 + myPort) para receber msgs de disseminacao
set dissPort [expr 8000 + $myPort]
#puts stderr "$myIP :: Socket de serverDiss na porta: $dissPort"
set err [catch {socket -server serverDiss $dissPort} msg]
if {$err != 0} {
    global myIP
    #puts stderr "$myIP :: Erro abrindo socket na porta $dissPort: $msg"
}
```

```
## - - - - - - - - - -

set loop 0

## Inicializa estruturas de posse, buffers, fds e portas de tokens
initToken
## Associa funcao de Recebimento de tokens ao evento de chegada de um token
initRecebeToken

set proceed false
after 5000 {set proceed true}
vwait proceed

#################################
## Dispara os testes:
#################################

foreach no $myNeigh {
    tokenTest $no
}
```