LUIS CARLOS ERPEN DE BONA

GERÊNCIA CONFIÁVEL DE REDES LOCAIS BASEADA EM DIAGNÓSTICO DISTRIBUÍDO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Elias P. Duarte Jr.

CURITIBA
2002

UFPR
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 *Luis Carlos Erpen de Bona*, avaliamos o trabalho intitulado, *"Gerência Confiável de Redes Locais Baseada em Diagnóstico Distribuído"*, cuja defesa foi realizada no dia 01 de março 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, 01 de março de 2002.

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

Prof. Dr. Francisco Vilar Brasileiro
DSC/UEPB

Prof. Dr. André Luiz Pires Guedes
DINF/UFPR

Sumário

Lista de Figuras

Resumo

Este trabalho apresenta uma ferramenta para gerência confiável de redes baseada em diagnóstico distribuído. A ferramenta utiliza os serviços do protocolo de gerência SNMP (*Simple Network Management Protocol*) e implementa o algoritmo *Hi-ADSD with Timestamps*. A ferramenta é composta por um sistema onde múltiplos agentes SNMP, executando o algoritmo de diagnóstico distribuído, monitoram a si mesmos e a um conjunto de serviços e dispositivos da rede. A estratégia para garantir a monitoração destes serviços e dispositivos de redes no algoritmo *Hi-ADSD with Timestamps* é uma contribuição deste trabalho. A MIB (*Management Information Base*) chamada `Test-MIB` é especificada neste trabalho, esta MIB é implementada por cada unidade da ferramenta e fornece as estruturas utilizadas pelo algoritmo de diagnóstico. A `Test-MIB` também permite a definição de procedimentos de testes específicos para os elementos de rede monitorados. As informações obtidas pela ferramenta são disponíveis através da MIB, que pode ser acessada através de aplicações SNMP ou através de uma interface Web. Resultados práticos apresentados incluem a utilização da ferramenta para monitorar uma rede local. Um servidor Web tolerante a falhas usando a ferramenta de diagóstico também é apresentado.

Abstract

This work presents a dependable network management tool based on distributed diagnosis. The tool is based on SNMP (*Simple Network Management Protocol* and implements the *Hi-ADSD with Timestamps* algorithm. The tool is composed by multiple SNMP agents that run the diagnosis algorithm, monitoring themselves and a configurable set of network services and devices. The strategy to allow network entities that cannot run the algorithm to be monitored is one of the contribution of this work. An MIB (*Management Information Base*) is specified and implements the data structures employed by the diagnosis algorithm. The MIB also allows the definition of test procedures tailored for each network element monitored. The information obtained by the tool is available on the MIB, that can be accessed by SNMP applications or a Web based interface. Practical results are described, the tool has been used for several months to monitor a local area network with over 60 machines, A fault-tolerant Web server built on top of the tool is also presented.

Capítulo 1

Introdução

As redes de computadores têm se tornando cada vez maiores e mais complexas, interligando computadores e componentes dos mais diversos tipos. A gerência automatizada destes sistemas é uma tarefa complexa e importante. Uma das atividades dos sistemas de gerência é determinar e descobrir as causas, se possível, das falhas que ocorrem na rede. É fundamental que esta tarefa seja desempenhada de forma confiável, ou seja, continue funcionando mesmo quando ocorrem falhas no sistema em que ela é executada. Este trabalho apresenta uma abordagem para gerência de rede confiável baseada em diagnóstico distribuído.

Este capítulo é organizado da seguinte forma. Primeiro são apresentados os sistemas de gerência de rede e sua funcionalidade. Então, os algoritmos e conceitos de diagnóstico distribuído são introduzidos. Em seguida é apre-

sentada uma ferramenta de gerência de redes baseada em um algoritmo de diagnóstico distribuído.

1.1 Sistemas de Gerência de Redes

Os sistemas de gerência de rede disponibilizam ferramentas que permitem controlar e monitorar as redes de computadores. O *Simple Network Management Protocol version 3* (SNMPv3) é a arquitetura de gerência padrão da Internet. Um sistema de gerência SNMPv3 é composto por entidades de gerência que se comunicam usando o protocolo de gerência. A arquitetura de gerência também define uma Base de Informações de Gerência ou *Managament Information Base* (MIB) que oferece uma interface para um conjunto de objetos de gerência através dos quais as aplicações de gerência podem monitorar e controlar entidades gerenciadas [1].

Tradicionalmente as entidades do SNMPv3 são chamadas gerentes e agentes. Cada elemento gerenciado possui um agente que mantém a sua MIB. Todo sistema de gerência possui pelo menos uma estação de gerência de rede ou *Network Management Station* (NMS), que executa ao menos uma entidade gerente. Gerentes são coleções de aplicações em nível de usuário que podem ser voltadas para avaliação de desempenho, diagnóstico de falhas entre outras aplicações. Atualmente existe um grande número de sistemas baseados

em SNMP, tanto comerciais quanto de domínio público.

Um dos componentes mais importantes em um sistema de gerência é o sub-sistema de gerência de falhas. O objetivo do sistema de gerência de falhas é descobrir, isolar e solucionar rapidamente as falhas que ocorrem na rede [2]. É essencial que um sistema de gerência de falhas seja tolerante a falhas, ou seja, capaz de funcionar corretamente mesmo na presença de falhas na rede em que ele é executado [3]. Atualmente, entretanto, a maioria dos sistemas de gerência de falhas não é tolerante a falhas [4]. Na medida em que o paradigma tradicional gerente-agente é substituído por uma abordagem distribuída, um novo foco em confiabilidade e desempenho aparece para sistemas de gerência baseados em SNMP [5].

1.2 Diagnóstico Distribuído

Considere um sistema composto por N nodos, que podem estar falhos ou sem-falhas. Assuma que o sistema é totalmente conectado, portanto existe um enlaces entre quaisquer pares de nodos e que os enlace não falham. Esta abordagem é usada freqüentemente para modelar redes locais em que todos os nodos compartilham o mesmo meio de comunicação. Um algoritmo de diagnóstico distribuído em nível de sistema permite que os nodos sem-falha determinem o estado de todos os outros nodos do sistema. Os nodos sem-

falhas são capazes de executar e informar os testes de maneira confiável [6, 7, 8]. Um algoritmo de diagnóstico é chamado adaptativo se os nodos determinam os próximos testes a serem executados com base nos resultados dos testes anteriores [9].

Uma das metas do diagnóstico em nível de sistema é que os nodos sem-falhas realizem a tarefa de diagnóstico no menor intervalo de tempo possível. Este intervalo de tempo é conhecido como a *latência* do algoritmo. A latência geralmente é dada em termos de rodadas de testes. Uma *rodada de teste* é definida como o tempo necessário para que todos os testadores encontrem ao menos um nodo sem-falhas. Um algoritmo de diagnóstico em nível de sistema também deve minimizar o número de testes que um nodo executa e o número de testes recebidos em cada nodo. Além destes fatores ainda é desejável que a quantidade de informações de diagnóstico trocada entre os nodos seja minimizada.

O algoritmo *Hierarchical Adaptive Distributed System-Level Diagnosis with Timestamps* (*Hi-ADSD with Timestamps*) foi introduzido em [10]. O *Hi-ADSD with Timestamps* utiliza uma estratégia de testes baseadas em *clusters*. Em um sistema com N nodos, um cluster é um conjunto de nodos testados a cada intervalo de testes, no *Hi-ADSD with Timestamps* os clusters têm sempre $N/2$ nodos. Quando um nodo sem-falhas testa um outro nodo sem-falhas de um determinado cluster, são obtidas as informações so-

bre todos os nodos deste cluster. Considere um nodo sem-falhas rodando o algoritmo, a figura 1.1 mostra os clusters testados pelo nodo 0. Quando o nodo 0 testa o nodo 1 este obtém informações sobre 4 nodos ($N/2$ nodos), ou seja, nodo 1, nodo 3, nodo 5 e nodo 7. Os outros dois clusters são análogos.

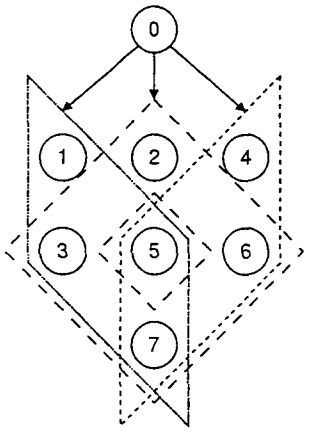

Figura 1.1: Clusters testados pelo nodo 0.

Cada nodo que executa o algoritmo guarda um *timestamp* para o estado de cada nodo do sistema. O *timestamp* é implementado como um contador incrementado a cada mudança de estado do nodo [11, 12]. Desta maneira, cada testador pode obter informações sobre um nodo do sistema a partir de mais de um nodo sem causar inconsistências. O uso de *timestamps* evita que informação desatualizada sobre um estado de um nodo seja confundida com informação atualizada.

Os nodos executando o algoritmo são capazes de diagnosticar eventos dinâmicos desde que um nodo permaneça em um dado estado por um período

de tempo suficiente para que todos os seus testadores detectem aquele estado. Apesar do custo para manter e transferir os *timestamps*, o algoritmo *Hi-ADSD with Timestamps* reduz de forma significativa a latência em relação a outras alternativas, apresentando uma nova opção prática para implementação de diagnóstico [10]. Foi mostrado que o *Hi-ADSD with Timestamps* permite que todos os nodos sem-falhas de um sistema completem o diagnóstico em um período de tempo menor do que de outros algoritmos similares.

Elementos da rede como impressoras ou switches podem não ser capazes de executar o algoritmo de diagnóstico distribuído. Os testes ditos secundários são introduzidos para permitir o monitoramento destes elementos. A definição de uma estratégia para a realização de testes secundários no algoritmo *Hi-ADSD with Timestamps* é uma contribuição deste trabalho. Para cada nodo que executa o algoritmo, chamado de agora em diante de nodo primário, é atribuído um conjunto de testes secundários. A cada intervalo de testes, cada nodo primário executa além dos testes definidos pelo algoritmo os testes secundários que lhe são atribuídos.

As informações sobre os resultados dos testes secundários são propagadas junto com as demais informações de diagnóstico. Quando um nodo primário falha ele deixa de executar os testes secundários, estes testes são atribuídos para um outro nodo primário sem-falhas. Considere o anel lógico formado

pelos nodos primários de acordo com seus identificadores seqüênciais, tal que o nodo i precede o nodo $i+1$ no anel; e o nodo $N-1$ precede o nodo 0. Os testes secundários de um nodo falho são assumidos pelo nodo sem-falhas que o precede no anel. A figura 1.2 mostra um exemplo desta estratégia. Os nodos rotulados de 0 até 3 são nodos primários que executam testes pré-definidos nos nodos secundários rotulados de a até h. Como o nodo 1 está falho o nodo 0 assume os testes secundários atribuídos ao nodo 1. Assim o nodo 0 passa também a testar os nodos c e d.

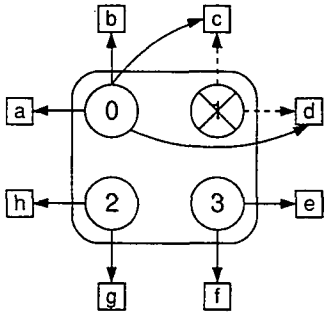

Figura 1.2: Nodos primários executando testes secundários.

1.3 Gerência de Redes Baseada em Diagnóstico Distribuído

Este trabalho apresenta uma ferramenta de gerência de redes distribuída e confiável na qual múltiplos agentes SNMP executando o algoritmo *Hi-ADSD*

with Timestamps [10], monitoram a si mesmos e a um conjunto configurável de serviços e dispositivos de rede. O sistema é confiável no sentindo em que continua funcionando mesmo se apenas um agente estiver sem-falhas.

Cada agente contém uma MIB que permite a especificação de testes de serviços e dispositivos de rede. Um teste é definido por um procedimento concebido especificamente para o componente testado. A freqüência em que cada teste é executado também pode ser especificada. O sistema também apresenta uma interface Web configurável, o que permite ao gerente humano monitorar a rede a partir de qualquer agente.

Resultados práticos são apresentados incluindo os resultados da monitoração distribuída de 67 máquinas ligadas por uma rede local e um servidor Web tolerante a falhas utilizando a ferramenta de diagnóstico. O servidor Web se manteve acessível em 97.35% do tempo em que foi observado, em um ambiente onde os servidores Web ficaram disponível apenas entre 13% e 66% do tempo.

1.4 Organização do Trabalho

O restante deste trabalho está organizado da seguinte maneira. O capítulo 2 apresenta conceitos e algoritmos de diagnóstico em nível de sistema. O capítulo 3 descreve um novo modelo em SNMP para sistema distribuído de

testes. O capítulo 4 apresenta uma ferramenta desenvolvida para gerência confiável de redes locais bem como aplicações práticas da ferramenta. Por fim, a conclusão é apresentada no capítulo 5.

Capítulo 2

Diagnóstico Distribuído em Redes Locais

Considere um sistema com N nodos. Assuma que o sistema é totalmente conectado, ou seja, existe um enlace de comunicação entre qualquer par de nodos, considere ainda que os nodos podem estar falhos ou sem-falhas e que os enlaces de comunicação não falham. O objetivo do diagnóstico distribuído em nível de sistema é permitir que todos os nodos sem-falhas determinem o estado, falho ou sem-falhas, de todos os nodos do sistema. Os nodos em um sistema de diagnóstico são capazes de testar outros nodos e determinar seu estado corretamente. Utilizando algoritmos de diagnóstico distribuído é possível construir sistemas de monitoração de rede tolerante a falhas [13]. Este capítulo apresenta conceitos e algoritmos de diagnóstico em nível de

sistema.

2.1 O Modelo PMC

O modelo PMC foi proposto por Preparata, Metze e Chien, daí o nome PMC [6]. O modelo PMC utiliza a teoria de grafos em seu formalismo, os vértices representam unidades do sistema capazes de testar outras unidades, ou nodos, e as arestas direcionadas indicam os testes realizados. Uma aresta saindo do nodo u para o nodo v significa que o nodo u testa o nodo v. Cada teste corresponde a um estímulo enviado cuja resposta será classificada em *pass* ou *fail* mapeados respectivamente para 0 e 1 nos pesos das arestas. O conjunto destes resultados é definido como sendo a síndrome do sistema.

Para determinar o estado de cada nodo do sistema é necessário analisar a síndrome, onde resume-se a tarefa de diagnóstico. Nem sempre a existência de um teste com o resultado *fail* significa que o nodo testado é falho. Isto porque o resultado de um teste só deve ser considerado caso o nodo testador seja confiável. Se o nodo testador for falho o resultado do teste não é previsível e não informa nada a respeito do nodo testado.

No modelo PMC não são consideradas falhas de enlaces e o sistema é representado por um grafo completo. Os testes realizados são fixados previamente e o conjunto destes testes determina se um sistema pode ou não ser

diagnosticado. Em [6, 7] são examinadas as condições necessárias e suficientes para que um conjunto de testes permita obter o diagnóstico correto do sistema.

Um sistema é dito *t-diagnosticável em um passo* quando é possível identificar todas as unidades falhas através da análise da síndrome. Para que um sistema com N nodos seja t-diagnosticável em um passo, o número de unidades falhas, t, não pode ultrapassar $\frac{N-1}{2}$. Além disto cada unidade deve ser testada por pelo menos t outras unidades. Então num sistema com $N = 2t+1$ são necessários $N * \frac{N-1}{2}$ testes, sendo que o número de testes requeridos é da ordem de $O(N^2)$.

A estratégia de fixar previamente todos os testes pode dificultar o diagnóstico, pois é preciso atribuir antecipadamente testes que compensem os testes que deixam de ser executados pelas unidades com falhas. Desta forma, foram criados os *algoritmos adaptativos* [9] onde os testes não são fixos, sendo determinados de acordo com os resultados dos testes realizados anteriormente.

Outra característica do modelo PMC é a necessidade de um nodo que concentre os resultados dos testes e realize o diagnóstico. A idéia de eliminar esta unidade distribuindo a tarefa de diagnóstico entre os nodos que realizam os testes é que fez surgir os *algoritmos distribuídos* [8].

2.2 O Algoritmo *Adaptive-DSD*

O *Adaptive-DSD* (*ADSD*) [13, 14] é um algoritmo de diagnóstico em nível de sistema ao mesmo tempo distribuído e adaptativo criado por Bianchini e Buskens. Por ser distribuído, não existe uma unidade central para realização do diagnóstico. Por ser adaptativo, a configuração dos testes vai alterando-se durante a execução do algoritmo.

Os algoritmos adaptativos possuem a noção de *rodadas de testes* (*testing rounds*), que é o período de tempo necessário para que todos os nodos executem os testes que lhe foram atribuídos. No algoritmo *ADSD* a rodada de testes é definida como o tempo necessário para que todos os nodos sem-falha do sistema testem pelo menos um nodo sem-falhas. A *latência* é definida como o número de rodadas de testes necessárias para que todos os nodos do sistema realizem o diagnóstico de um evento ocorrido. Um *evento* é a transição de um nodo do estado de sem-falhas para falho ou do estado de falho para sem-falhas. O número de testes realizados a cada rodada de testes e a latência são medidas utilizadas para comparar a eficiência de algoritmos de diagnóstico adaptativos.

No algoritmo *ADSD* cada nodo testa outros nodos até encontrar um nodo sem-falhas do qual colhe informações para atualizar suas informações de diagnóstico. Uma conseqüência da estratégia de testes do *ADSD* é que cada nodo

é testado por um único nodo sem-falhas a cada rodada.

Em um sistema com N nodos executando o algoritmo *ADSD*, os nodos são identificados como u_1, u_2,...,u_N. O nodo u_i também é chamado nodo i. A adição usada é módulo N, de modo que o nodo u_N é seguido por $u_N + 1$ que é u_0, formando um anel de nodos. Os testes são realizados seqüencialmente, assim u_i testa u_{i+1} e seguintes até encontrar um nodo sem-falhas, então atualiza sua informação local de diagnóstico com as informações lidas do nodo testado. Estas informações são resultantes tanto de testes realizados pelo nodo testado como de informações lidas de outros nodos.

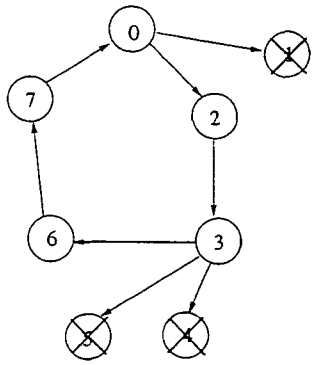

Figura 2.1: Sistema executando o algoritmo *ADSD*.

A figura 2.1 mostra um sistema executando o algoritmo *ADSD*. Os nodos assinalados estão falhos e os não assinalados sem-falhas. Em uma rodada de testes cada nodo testa seqüencialmente seus sucesssores. O nodo 0, por exemplo, testa o nodo 1, descobrindo que o nodo 1 está falho, então testa o nodo 2, sem-falhas, lendo as informações de diagnóstico dele. De forma

análoga comportam-se os nodos 2, 3, 6 e 7.

Em termos de quantidade de testes a cada rodada o algoritmo *ADSD* é considerado eficiente sendo executados no máximo N testes a cada rodada. Quanto à latência, podem ser necessárias até N rodadas para que todos os nodos realizem o diagnóstico de um evento. Isto ocorre porque a informação sobre o estado de um nodo é passada para apenas um novo nodo a cada rodada. Imagine que a cada rodada os testes estão sendo feitos seqüencialmente por u_0, u_1, etc, se o nodo u_N falhar o nodo u_{N-1} detecta a falha, mas apenas na outra rodada o nodo u_{N-2} lê informações de u_{N-1} e descobre a falha em u_N, da mesma forma ocorre com os demais nodos. Com isto, pode ser preciso N rodadas de testes até a informação da falha de u_N chegar a u_0.

2.3 O Algoritmo *Hi-ADSD*

O algoritmo *Hierarchical Adaptive Distributed System-Level Diagnosis* (*Hi-ADSD*) [15] representa uma evolução do algoritmo ADSD, melhorando substancialmente sua latência. Além de distribuído e adaptativo, o algoritmo *Hi-ADSD* também é hierárquico, realizando testes baseados numa estratégia de dividir para conquistar.

A hierarquização dos testes é alcançada utilizando o conceito de *clusters*. Os *clusters* são conjuntos de nodos testados a cada rodada, o tamanho

aumenta progressivamente a cada rodada sempre sendo uma potência de dois. Um *cluster* de p nodos $n_j, ..., n_{j+p-1}$ onde $j\ MOD\ p = 0$ e p é uma potência de dois, é definido recursivamente como um nodo, quando $p = 1$; ou como a união de dois *clusters*, um contendo os nodos $n_j, ..., n_{j+p/2-1}$ e outro $n_{j+p/2}, ..., n_{j+p-1}$. A figura 2.2 mostra um sistema de oito nodos organizado em *clusters*.

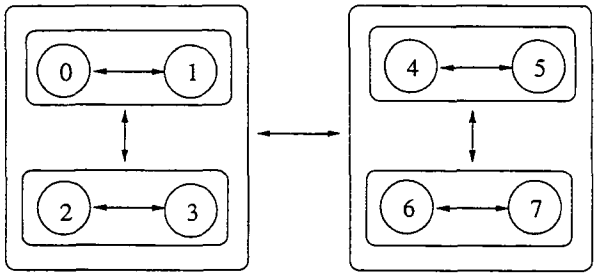

Figura 2.2: Sistema de 8 nodos agrupados em clusters no algoritmo *Hi-ADSD*.

Um nodo sem-falhas inicialmente realiza testes em *clusters* de tamanho 1 (2^0), nas próximas rodadas os *clusters* têm tamanho 2 (2^1), 4 (2^2), 8 (2^3) até atingir $N/2$ ($2^{(log_2 N)-1}$) nodos. A lista ordenada de nodos testados por um determinado nodo i em um cluster de tamanho 2^{s-1} é dada por $C_{i,s}$. A expressão que caracteriza $C_{i,s}$ é dada como segue:

$$C_{i,s} = \begin{array}{l} \{n_t | t = (i\ MOD\ 2^s + 2^{s-1} + j)\ MOD\ 2^{s-1+a} + \\ (i\ DIV\ 2^s) * 2^s + b * 2^{s-1}; j = 0, 1, ..., 2^{s-1} - 1\}, \end{array}$$

onde

$$a = \begin{cases} 1 & se\ MOD\ 2^s < 2^{s-1} \\ 0 & outros\ casos \end{cases} \qquad b = \begin{cases} 1 & se\ a = 1\ AND\ (i\ MOD\ 2^s + 2^{s-1} + j) \\ & MOD\ 2^{s-1+a} + (i\ DIV\ 2^s) * 2^s < i \\ 0 & outros\ casos \end{cases}$$

Quando um nodo i realiza testes em nodos pertencentes a $C_{i,s}$, ele executa testes seqüencialmente até encontrar um nodo sem-falhas, ou até descobrir que todos os outros nodos estão falhos. Pela função $C_{i,s}$ quando dois nodos diferentes testam nodos de um *cluster* eles iniciam testando nodos diferentes. A tabela 2.1 mostra a função $C_{i,s}$ para um sistema com 8 nodos.

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

Tabela 2.1: $C_{i,s}$ para um Sistema com 8 Nodos.

Quando um nodo sem-falhas é testado ele fornece informações sobre todos os nodos de $C_{i,s}$ ao qual ele pertence. Se nenhum nodo sem-falhas é encontrado em um determinado $C_{i,s}$ o nodo i passa a testar $C_{i,s+1}$ no mesmo intervalo de testes, até encontrar um nodo livre de falhas, ou descobrir que todos os nodos estão falhos. A figura 2.3 mostra os *clusters* testados pelo nodo 0. Inicialmente o nodo 0 testa o nodo 1 do *cluster* de tamanho 1, na próxima rodada de testes testa o *cluster* de tamanho 2 composto pelos nodos

2 e 3, por fim testa o *cluster* de tamanho 4 que contém os nodos 4, 5, 6 e 7 e então retorna para testar o *cluster* de tamanho 0.

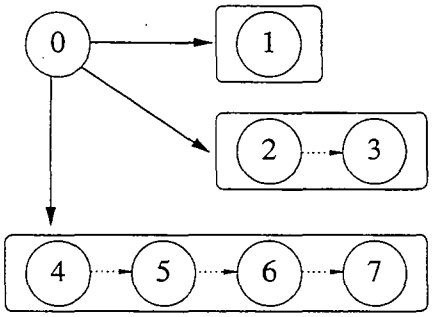

Figura 2.3: Clusters testados pelo nodo 0 executando o algoritmo *Hi-ADSD*.

O algoritmo realiza testes de forma assíncrona, isto significa que em uma mesma rodada os nodos podem estar testando *clusters* de tamanhos diferentes. A latência pode chegar a, no máximo, $(log_2 N)^2$ rodadas de testes. Simulações [15] demonstraram que a latência em geral é menor que $(log_2 N)^2$, e que se os nodos estiverem realizando os testes de forma sincronizada a latência é da ordem de $log_2 N$ rodadas de teste.

No algoritmo *Hi-ADSD* em uma situação especial podem ser executados $N^2/4$ testes em uma única rodada. Suponha que exista um *cluster* com tamanho $N/2$ onde todos os nodos estão falhos e que os $N/2$ nodos restantes e sem-falhas realizem testes sobre este *cluster* falho. Então teremos $N/2$ nodos realizandos $N/2$ cada, resultando em um total de $N^2/4$ testes. Isto representa um problema do algoritmo já que neste caso o número de testes

executados em um rodada pode ser da ordem $O(N^2)$, que é a mesma ordem de complexidade de algoritmos de força bruta onde todos os nodos testam todos os nodos.

2.4 O Algoritmo *Hi-ADSD with Detours*

Com o objetivo de diminuir o número máximo de testes realizados no algoritmo *Hi-ADSD* surgiu o algoritmo *Hi-ADSD with Detours* [16]. O objetivo é cumprido por meio de uma mudança na estratégia de testes. O resultado é que a cada $log_2 N$ rodadas são executados no máximo $N log_2 N$ testes. Os *clusters* são organizados da mesma maneira que no algoritmo *Hi-ADSD* e a latência também é mantida.

No algoritmo *Hi-ADSD with Detours* quando o nodo testa um nodo falho de um determinado *cluster* ele não continua a testar os demais nodos deste *cluster*. Nesta situação o *cluster* é dito *bloqueado* e o nodo testador procura obter informações sobre os nodos deste *cluster* ao testar outros *clusters*. As informações podem ser obtidas por caminhos alternativos no grafo de testes. Estes caminhos alternativos são chamados de *detours*. Testes extras são realizados nos nodos do *cluster* bloqueado para os quais não foram encontrados *detours*.

Os nodos do algoritmo *Hi-ADSD with Detours* são organizados em *clus-*

ters da mesma forma que no algoritmo *Hi-ADSD*. A definição de $C_{i,s}$ também é a mesma adotada no *Hi-ADSD* original, apresentada na seção anterior.

Uma *rodada de testes* é definida como o período de tempo no qual todos os nodos sem-falhas testam um *cluster*. A latência do algoritmo é definida como o número de rodadas de testes necessárias para que todos os nodos sem-falha do sistema realizem o diagnóstico de um evento. Considera-se que durante o diagnóstico de um evento um novo evento não ocorre no sistema.

O grafo de *testes livres de falhas do sistema*, $T(S)$, é um grafo onde os vértices são os nodos do sistema e uma aresta direcionada entre o nodo i e j indica que o nodo i testou o nodo j como sem-falhas no último intervalo de teste. Quando todos os nodos estão sem-falhas $T(S)$ é um hipercubo.

A *distância de diagnóstico* entre o nodo i e o nodo p, $d_{i,p}$, é a menor distância entre os nodos i e p em $T(S)$. Por exemplo, na figura 2.4, podemos verificar que $d_{0,5} = 2$, ou seja a distância de diagnóstico do nodo 0 até o nodo 5 é 2. Ainda define-se $D_{i,k}$ como sendo o conjunto de todos os nodos p tal que $d_{i,p} <= k$. Assim de acordo com a figura 2.4, $D_{0,1} = \{1, 2, 4\}$, $D_{0,2} = \{1, 2, 4, 3, 5, 6\}$ e $D_{0,3} = \{1, 2, 4, 3, 5, 6, 7\}$.

$R_{i,s,p}$ é a lista ordenada de nodos que podem ser alcançadas pelo nodo i a partir do nodo p com distância de diagnóstico menor ou igual a s e menor que a distância do nodo i quando todos os nodos estão sem-falhas. $R_{i,s,p}$ é dada pela expressão abaixo. A figura 2.4 mostra $R_{0,3,2} = \{6, 7\}$.

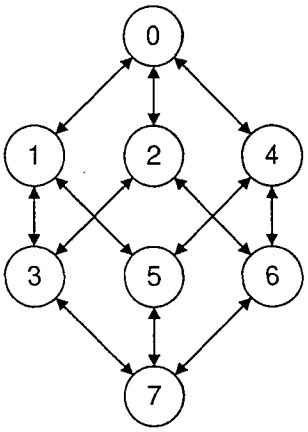

Figura 2.4: O grafo $T(S)$ para um sistema com 8 nodos.

$$R_{i,s,p} = (C_{i,s} \cap D_{p,s-1}) - D_{i,d_{i,p}}$$

Um *detour* do nodo i ao nodo j é um caminho em $T(S)$ do nodo i ao nodo j passando por nodos fora do *cluster* ao qual o nodo j pertence. O *detour* deve ter exatamente o mesmo número de arestas do caminho mais curto entre i e j no caminho que contém apenas nodos do *cluster* ao qual j pertence.

A figura 2.5 mostra um sistema com 8 nodos executando o algoritmo *Hi-ADSD with Detours*, os nodos assinalados estão falhos. As linhas pontilhadas mostram os testes que seriam executados pelo *Hi-ADSD* sem *detours* para diagnosticar os nodos 5, 6 e 7. No *Hi-ADSD with Detours* a informação do nodo 5 é obtida através do nodo 1, e as informações sobre os nodos 6 e 7 são obtidas através do nodo 2.

A especificação do algoritmo é feita utilizando as funções more-info e more-

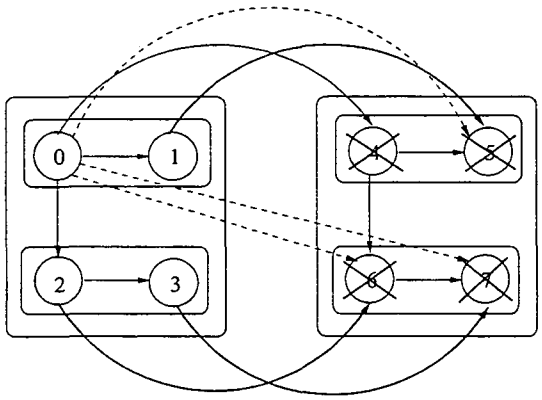

Figura 2.5: Sistema executando o algoritmo *Hi-ADSD with Detours.*

tests. O testador executa more-info depois de testar um nodo sem-falhas para determinar se é necessário obter informações adicionais pelo nodo testado. A função é dada abaixo, onde: nodo i é o testador; nodo p é o nodo sem-falhas testado em $C_{i,s'}$; a lista $R_{i,log_2N,p}$ contém todos os nodos cujas informações podem ser obtidas pelo nodo i através do nodo p; e o conjunto $B_{i,s}$ contém os nodos bloqueados de um dado $C_{i,s}$. Quando o nodo testada em um dado *cluster* está falho e o testador não obtém informações sobre os outros nodos do *cluster* estes nodos são chamados de bloqueados.

$$\text{more-info}(i,p) = R_{i,log_2N,p} \cap B_{i,s}, s' < s <= log_2N$$

A função more-tests é executada quando um nodo falho é testado, para determinar se é necessário executar testes adicionais no mesmo *cluster*, ou seja, se não existem *detours* para os nodos bloqueados. A função more-tests é dada pela fórmula abaixo, onde o nodo i é o testador, o nodo p é o nodo

testado falho, $C_{i,s,p}$ a lista contendo todos nodos sobre os quais o nodo p pode dar informação para o nodo i e s varia de acordo com o $C_{i,s}$ ao qual p pertence:

$$\text{more-testes}(i,s) = B_{i,s} - R_{i,s,p} | p \in C_{i,s'} \; e \; 1 <= s' < s$$

O pseudo código é dado abaixo.

```
ALGORITHM Hi-ASD with Detours {at node i}
  initializa all sets Bi,s empty;
  REPEAT FOREVER
    FOR s = 1 to LogN DO
      p := first node in Ci,s
      test(p)
      IF p is fault-free THEN
         get Ci,s diagnosis information;
         update Bi,s;
         get more_info(i,p) diagnosis information;
         update Bi,s;
      ELSE p is faulty THEN
         Bi,s = Ci,s - p
         WHILE more-tests(i,s) is not empty DO
           k = first node in more_tests(i,s)
           test(k); Bi,s = Bi,s - k;
           IF k is fault-free THEN
                get Bi,s diagnosis information; update Bi,s;
                get more_info(i,k) diagnosis information; update Bi,s;
           END IF
         END WHILE
      END IF
    END FOR
  END REPEAT
```

Simulações mostram que o *Hi-ADSD with Detours* [16] precisa de um número menor de testes em relação ao *Hi-ADSD* original em diversas situações de falhas. Na tabela 2.2 estão os resultados da simulação de um sistema com 64 nodos. Os resultados mostram que o emprego de *detours*

Situação de Falha	*Hi-ADSD*	*Hi-ADSD with Detours*
32 nodos falhos de forma aleatória	383	285
Pior caso do Hi-ADSD	1184	191
Pior caso do Hi-ADSD with Detours	384	384

Tabela 2.2: Simulações de um sistema com 64 nodos executando os algoritmos Hi-ADSD e Hi-ADSD com detours

diminui o número de testes necessários para completar o diagnóstico do sistema.

2.5 O Algoritmo *Hi-ADSD with Timestamps*

Eventos são definidos como a mudança de estado de um nodo detectada pelo algoritmo de diagnóstico. Um evento ocorre quando um nodo falho se torna sem-falhas ou quando um nodo sem-falhas se torna falho. Os algoritmos hierárquicos baseados no modelo PMC assumem situações de falhas estáticas, não podendo haver um novo evento enquanto todos os nodos sem-falhas do sistema não detectarem o evento anterior. Esta asserção é artificial e pode não ser garantida facilmente em um sistema real.

O algoritmo *Hi-ADSD with Timestamps* [10] é um algoritmo baseado

no algoritmo *Hi-ADSD with Detours* e trabalha com falhas dinâmicas, onde eventos podem ocorrer antes do diagnóstico do evento anterior ter sido completado por todos os nodos. O *timestamp* é uma informação guardada sobre o estado de cada nodo do sistema, a informação é um contador incrementado a cada mudança de estado [11, 18]. A utilidade do *timestamp* é permitir datar informações, distinguindo as informações mais atualizadas das menos atualizadas.

Ao testar um nodo sem-falhas em um *cluster*, no algoritmo *Hi-ADSD with Timestamps*, não são obtidas somente informações sobre o *cluster* do nodo testado e possíveis *detours*. Sempre que um nodo i testa um nodo j como sem-falhas são lidas informações de diagnóstico de todos os nodos que estão a uma distância de diagnóstico de até $log_2N - 1$ do nodo i passando pelo nodo j, um conjunto que tem sempre $N/2$ nodos. Como as informações sobre um determinado nodo podem ser obtidas a partir de vários nodos testados, então emprega-se o *timestamp* para selecionar a informação de diagnóstico mais recente.

A latência média no *Hi-ADSD with Timestamps* é reduzida em relação ao *Hi-ADSD with Detours*, embora no pior caso o número de testes requeridos ainda seja $(log_2N)^2$. O número máximo de testes é o mesmo do *Hi-ADSD with Detours* já que a estratégia de testes é a mesma. Simulações feitas em sistemas de 512 nodos mostram que o algoritmo *Hi-ADSD with Timestamps*

[10] é em média quatro vezes mais rápido para completar o diagnóstico de uma falha que o *Hi-ADSD with Detours*. Para este sistema de 512 nodos o *Hi-ADSD with Detours* teve uma latência de 40 rodadas de teste, enquanto no *Hi-ADSD with Timestamps* a latência foi de 12 rodadas.

2.5.1 Diagnóstico Dinâmico de Eventos de Falha e Recuperação

Uma das melhorias do *Hi-ADSD with Timestamps* em relação aos outros algoritmos hierárquicos é a capacidade de diagnosticar certos eventos dinâmicos, ou seja, eventos que ocorrem antes do diagnóstico do evento anterior por todos os nodos sem-falha. O modelo de falhas estático, adotado pelos demais algoritmos hierárquicos de diagnóstico apresentados, considera que um novo evento não ocorre antes que todos os nodos sem-falhas tenham diagnosticado o evento anterior. O modelo estático considera que o algoritmo de diagnóstico está analisando uma imagem retirada do sistema em um determinado intervalo de tempo [19]. O modelo estático entretanto não é adequado para um ambiente real, uma vez que falhas podem ocorrer durante a execução do algoritmo de diagnóstico. Considerando-se um modelo de falhas dinâmico é necessária a especificação de um procedimento de recuperação para os nodos que são reparados durante a execução do algoritmo.

Quando um nodo falho é reparado, ele não tem informações de diagnóstico sobre o sistema. À medida em que o nodo começa a executar testes e obter informações de diagnóstico de outros nodos sem-falhas ele atualiza sua informação de diagnóstico. É necessário distinguir informações de diagnóstico atualizadas e não atualizadas. Caso contrário a informação incorreta, não atualizada, pode ser propagada pelo sistema.

Considere que as informações de diagnóstico que cada nodo mantém sobre os nodos do sistema são organizadas em uma tabela, onde cada linha corresponde a um nodo. Um campo chamado *u-bit* pode ser inserido em cada linha e utilizado para indicar se a informação de diagnóstico foi ou não atualizada desde a inicialização do nodo. Este campo pode ser implementado como um bit, cujo valor 1 indica que informação é atualizada e 0 em outro caso.

Ao reparar-se de uma falha o nodo atribui a todos os seus *u-bits* o valor 0. Ao encontrar um nodo sem-falhas que já tenha completado o processo de inicialização ele obtém informações de todos os nodos do sistema. Caso um testador se recupere e teste todos os nodos como falhos (ou não encontre nenhum nodo sem-falhas atualizado) então ele precisa atribuir todos *u-bits* para 1. Durante a inicialização do algoritmo ocorre algo similar, pois apesar de encontrar nodos sem-falhas estes nodos ainda não possuem informação de diagnóstico atualizadas, ou seja, têm o *u-bit* igual a 0.

2.5.2 Especificação do Algoritmo

Considere um sistema S composto de N nodos, que podem estar falhos ou sem-falhas. Assuma que o sistema é totalmente conectado, ou seja, existe um enlace de comunicação entre qualquer par de nodos, e os enlances não falham. Neste sistema os nodos sem-falhas são capazes de testar e comunicar os resultados de forma confiável.

O *intervalo de testes* é definido como o período de tempo no qual um nodo sem-falhas testa um *cluster*. O *grafo de testes livres de falhas do sistema*, $T(S)$, é um grafo direciondo cujos vértices representam os nodos do sistema e uma aresta direcionada entre o nodo i e j indica que o nodo i testou o nodo j como sem-falhas no último intervalo de teste. Quando todos os nodos estão sem-falhas $T(S)$ é um hipercubo. A *distância de diagnóstico* entre o nodo i e o nodo p, $d_{i,p}$, é a menor distância entre os nodos i e p em $T(S)$.

O *grafo de testes livres de falhas de i*, TFF_i, é um grafo direcionado cujos vértices são os nodos de S. Existe a aresta direcionada de a para b, ab, em TFF_i se ab pertence a $T(S)$ e $d_{i,a} < d_{i,b}$. A figura 2.6 mostra TFF_0 para um sistema com 8 nodos.

$C_{i,s,p}$ é definido como a lista ordenada de nodos que podem ser alcançados pelo nodo i a partir do nodo p com uma distância de diagnóstico menor ou igual a $s-1$. A figura 2.7 mostra $C_{0,3,2}$. A função $C_{i,s,p}$ corresponde ao

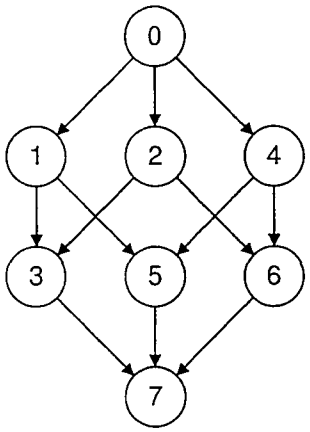

Figura 2.6: O grafo TFF_0 para um sistema com 8 nodos.

cluster testado quando o nodo i é o testador.

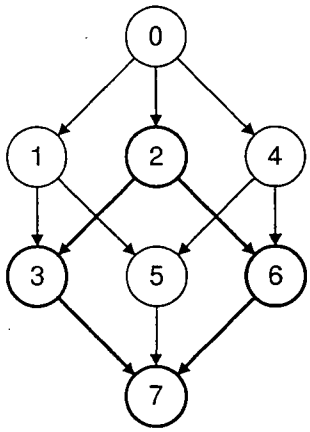

Figura 2.7: Um exemplo de $C_{i,s,p}$: $C_{0,3,2}$.

A estratégia de testes do algoritmo *Hi-ADSD with Timestamps* é a mesma utilizada pelo algoritmo *Hi-ADSD with Detours*. A cada intervalo de testes um nodo testa um cluster. Após $log_2 N$ intervalos de testes o primeiro *cluster* é testado novamente. A figura 2.8 mostra os *clusters* testados pelo nodo

0. Quando o nodo 0 testa o nodo 1 obtém informações sobre os nodos do cluster $C_{0,3,1}$, ou seja, nodo 1, nodo 3, nodo 5 e nodo 7. Os outros dois *clusters* testados, $C_{0,3,2}$ e $C_{0,3,4}$, são análogos.

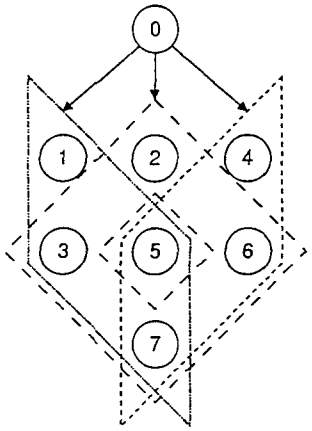

Figura 2.8: Clusters testados pelo nodo 0.

No *Hi-ADSD with Timestamps* sempre que um nodo sem-falhas é testado são obtidas informações do conjunto de nodos com distância de no máximo $log_2 N-1$, ou seja, sobre $N/2$ nodos. No *Hi-ADSD with Detours* o tamanho do conjunto de informações obtidas é limitado pelo tamanho do *cluster* testado mais os nodos retornados pela função more-info.

A limitação do conjunto de informações obtidas no *detours* é necessária porque no algoritmo *Hi-ADSD with Detours* um nodo i não pode obter informações de diagnóstico sobre um nodo j de dois ou mais nodos. No *Hi-ADSD with Timestamps* o nodo i pode obter informações sobre um nodo j de dois ou mais nodos sem-falhas. Neste caso é necessário determinar qual

informação sobre o estado de j é mais recente. Por exemplo, na figura 2.8, é possível verificar que o nodo 0 pode obter informações sobre o nodo 5 pelos *clusters* $C_{0,3,1}$ e $C_{0,3,4}$.

O *timestamp* é um mecanismo que permite datar a informação de diagnóstico e é implementado com um contador das mudanças de estado. Cada vez que um teste é executado e o testador descobre que o nodo testado mudou de estado o contador é incrementado.

Para garantir que o nodo i mantenha as informações mais recentes sobre nodo j ele precisa comparar o *timestamp* do estado de j obtido com o seu *timestamp* local. Se o timestamp do estado de j obtido é menor que o *timestamp* de j local então a informação de diagnóstico é desatualizada e i não deve atualizar o estado de j. Caso contrário, o *timestamp* local de j for menor do que o obtido a informação é mais nova, então o *timestamp* local e a informação de diagnóstico devem ser atualizadas.

Se o primeiro nodo testado de Ci, s, p estiver falho, o testador tentará encontrar *detours* para os nodos destes clustres através de outros nodos sem-falhas testados. Um teste extra somente é executado em *cluster* se não forem encontrado *detours*.

2.5.3 Testes Secundários

A especificação dos procedimentos correspondentes aos testes realizados pelas unidades do sistema é um dos principais problemas para implementar sistemas de diagnóstico. Geralmente um nodo a testa um nodo b comparando o resultado de uma entrada dada. Este teste, além de determinar corretamente o estado do nodo testado, deve ser executado de maneira rápida e sem consumir recursos excessivos.

Elementos de redes como impressoras ou *switches* podem não ser capazes de executar o algoritmo de diagnóstico distribuído. Os testes secundários são introduzidos para permitir o monitoramento destes elementos. Para cada nodo que executa o algoritmo de diagnóstico, agora chamado nodo primário, é atribuído um conjunto de testes secundários. A cada intervalo, cada nodo primário, além de executar os testes determinados pelo algoritmo de diagnóstico, executa os testes secundários que lhe foram atribuídos.

O teste secundário deve ser projetado especificamente para o elemento testado, pois não seria possível adotar um procedimento de teste padrão para elementos de redes diversos. Ainda, para atender as características de cada elemento testado, o intervalo de testes dos testes secundários pode ser diferente do intervalo de teste primário. Já que um determinado elemento de rede possua uma taxa de falha muito baixa ou que o custo para determinar

seu estado seja muito alto.

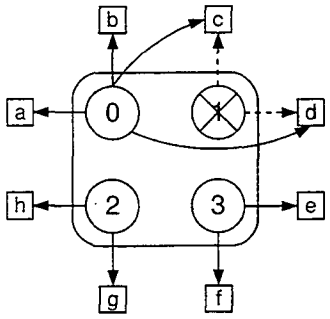

Figura 2.9: Nodos primários executando testes secundários.

As informações sobre a saída dos testes secundários propagam-se juntamente com as informações de diagnóstico restantes, ou seja, quando um nodo obtém informações sobre o estado de um determinado nodo, também vai obter informações sobre os testes secundários realizados por aquele nodo. Quando um nodo primário se torna falho, os testes secundários que lhe foram atribuídos deixam de ser executados. É necessário que um outro nodo primário sem-falhas assuma a execução destes testes secundários. Um procedimento para outro nodo assumir estes testes secundários é definido a seguir.

Considere o anel lógico formado por todos nodos primários de acordo com seu identificadores seqüênciais, de forma que o nodo i preceda o nodo $i+1$ no anel; e que o nodo $N-1$ preceda o nodo 0. O nodo sem-falhas no anel que precede o nodo falho assume a execução de teste secundário do nodo falho. A figura 2.9 mostra um exemplo. Os nodos identificados de 0 a 3

são nodos primários que executam testes pré-definidos em nodos secundários identificados de a a h. Como o nodo primário 1 está falho, o nodo 0 assume a execução dos testes secundários do nodo 1, ou seja, o nodo 0 também vai testar c e d além dos testes que executava anteriormente. O algoritmo para determinar quais testes secundários cada nodo deve executar é dado abaixo.

```
SECONDARY TESTS AT NODEi
  X = (i+1) MOD N
  REPEAT WHILE DIAGNOSE_INFO[X] == FAULTY AND X<>i
    NODEi assumes NODEx secondary testes
    X = (X+1) MOD N
  END REPEAT
END
```

O capítulo 4 apresenta uma ferramenta que implementa o algoritmo *Hi-ADSD with Timestamps*, incluíndo a estratégia para execução de testes secundários apresentada acima.

Capítulo 3

Usando o Protocolo SNMP para Testes em Redes Locais

Este capítulo apresenta um modelo de diagnóstico distribuído de testes baseado no protocolo SNMP [20, 21]. O protocolo SNMP é introduzido na seção 3.1, a seção 3.2 descreve um framework SNMP para a execução de testes e diagnóstico distribuído.

3.1 O Protocolo SNMP

Com o aumento da complexidade e popularidade das redes de computadores os sistemas integrados de gerência tornam-se cada vez mais necessários. As ferramentas disponibilizadas pelos sistemas de gerência de rede permi-

tem controlar e monitorar os diversos componentes de uma rede. O *Simple Network Management Protocol* (SNMP) é a arquitetura de gerência padrão da Internet. Um sistema de gerência SNMP é composto por estações de gerência de rede, componentes gerenciados, informações de gerência e um protocolo de gerência. A arquitetura clássica de gerência de redes é mostrada na figura 3.1.

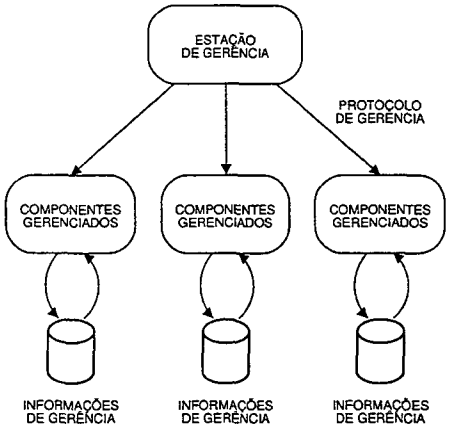

Figura 3.1: Arquitetura de gerência clássica.

Os componentes gerenciados são os *agentes*, e podem ser praticamente qualquer componente conectado à rede: computadores, protocolos, roteadores, terminais, impressoras, modems, *hubs* ou qualquer dispositivo que possa disponibilizar alguma informação à rede. O agente responde requisições de informações e ações da estação de gerência. Para o sistema de gerência os componentes gerenciados são representados por um conjunto de objetos. Cada objeto é essencialmente uma variável que representa um determinado

aspecto do componente gerenciado. A coleção destes objetos é chamada base de informação de gerência ou *Management Information Base* (MIB).

Todo sistema de gerência possui pelo menos uma Estação de Gerência de Rede ou *Network Management Station* (NMS), que executa ao menos uma aplicação *gerente*. Os gerentes são coleções de aplicações em nível de usuário que podem ser voltadas para avaliação de desempenho, diagnóstico de falhas entre outras aplicações. A tarefa de monitoração da rede é realizada pelos gerentes através da leitura de objetos das MIBs mantidas pelos agentes.

O SNMP utiliza a linguagem de definição abstrata ASN.1 (*Abstract Syntax Notation One*) para identificação e especificação dos objetos de gerência utilizados. O ASN.1 compreende uma linguagem de descrição de dados. Esta linguagem permite que sejam definidos tipos de objetos primitivos que podem ser combinados para obter objetos mais complexos. Cada tipo de objeto tem uma identificação e uma sintaxe de estrutura de dados e codificação. A sintaxe de codificação define como os tipos ASN.1 são codificados para a transmissão. Esta sintaxe evita ambigüidades permitindo aos diversos componentes enviarem informações sem problemas com a representação dos dados.

Os objetos de uma MIB são organizados de uma forma hierárquica, que pode ser representada por uma árvore. Na árvore os objetos são representados por nodos e as aresta são identificadas tanto por uma string quanto

por um número. A figura 3.2 mostra a organização hierárquica de uma MIB padrão. A identificação hierárquica de um objeto é chamada de *Object IDentifier* (OID). Um OID é dado pela lista de arestas percorridas da raiz até o objeto que se deseja indexar, por exemplo o identificador ".iso.org.dod.internet.mgmt.mib" está destacado na figura 3.2. O identificador ".iso.org.dod.internet.mgmt.mib" é usado como prefixo para outros grupos de objetos, cada qual com um identificador próprio. Por exemplo o grupo *system*, tem identificador ".iso.org.dod.internet.mgmt.mib.system". Dentro dos grupos ocorrem ramificações que levam até os objetos de gerência. Por exemplo, o objeto *sysDescr*, que é um dos objetos do grupo *system* é identificado como ".iso.org.dod.internet.mgmt.mib.system.sysDescr".

Segundo a linguagem ASN.1 as variáveis de uma MIB podem ser de dois tipos: variáveis simples e tabelas. As variáveis simples incluem tipos como inteiros, inteiros sem sinal e strings de caracteres. As tabelas correspondem a vetores de variáveis com uma dimensão. Por exemplo, uma tabela pode conter os discos rígidos de uma máquina; então a tabela vai conter uma entrada para cada disco. Se ao final do identificador o número for 0 (zero), trata-se de uma variável simples. Caso contrário, trata-se de uma tabela, onde o número identifica uma entrada na tabela.

O protocolo SNMP é usado para comunicação entre o gerente e o agente. O gerente pode solicitar informações da MIB de um agente, para isto en-

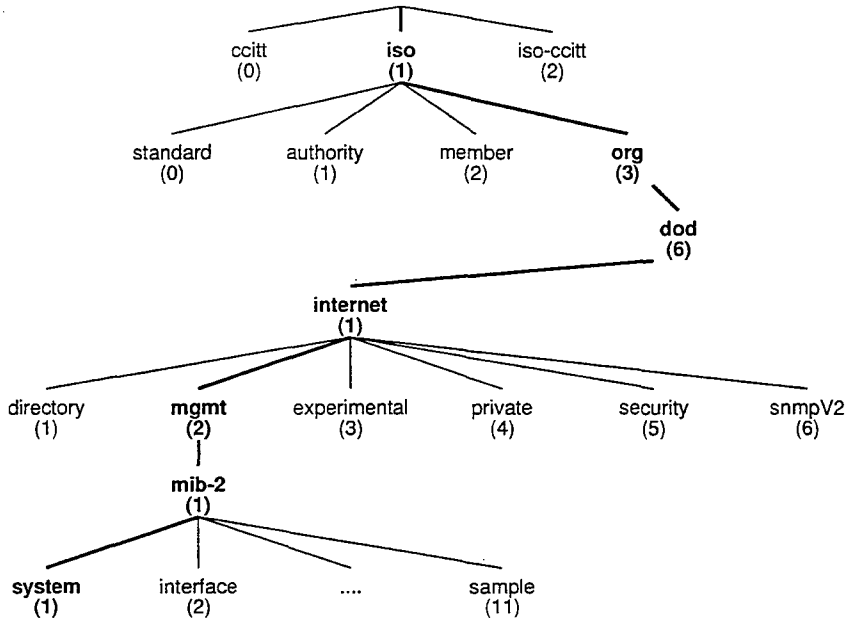

Figura 3.2: Organização hierárquica dos objetos de uma MIB.

viará mensagens próprias do protocolo. Nestas mensagens estão os objetos codificados através da sintaxe de codificação do ASN.1. O SNMP é um protocolo da camada de aplicação. As mensagens são representadas por um único datagrama UDP (*User Datagram Protocol*) usando as regras básicas de codificação do ASN.1. As mensagens definidas pelo protocolo podem ser classificadas em três categorias: leitura de objetos de gerência; modificação de valores de objetos e alarme. As mensagens da categoria de leitura (*get, get-next* e *get-bulk*) são usadas pelo gerente para requisitar um ou mais objetos da MIB de um agente. A mensagem *get* é usada para requisitar um ou mais objetos específicos. A mensagem *get-next* requisita o objeto com

o identificador seguinte ao informado na requisição. Por fim, a mensagem *get-bulk* é utilizada para requisitar e transferir de forma eficiente um grande número de objetos.

A categoria de modificação contém apenas a mensagem *set*. A mensagem *set* é usada para modificar objetos na MIB de um agente qualquer. Alguns objetos da MIB só podem ser lidos, não podendo ter seu valor modificado. As duas categorias de mensagens anteriores são enviadas pelo gerente para o agente. A mensagem da categoria de alarme *trap*, permite que um agente envie uma mensagem ao gerente informando a ocorrência de uma determinada condição anormal.

3.2 Um Modelo SNMP para Sistema Distribuído de Testes

O sistema distribuído de testes consiste em um conjunto de agentes que se comunicam utilizando SNMP. Cada agente, também chamado *testador*, mantém a `Test-MIB`, que é uma base de informação de gerência utilizada para manter informação sobre os testes executados pelo testador, assim como o estado de outros testadores e componentes do sistema.

A `Test-MIB` implementa as estruturas de dados do algoritmo de diagnós-

tico *Hi-ADSD with Timestamps*. Cada testador testa e obtém informações de outros testadores de acordo com o algoritmo de diagnóstico distribuído, estes testes são chamados *testes primários*. Além destes testes também são executados os chamados *testes secundários*. Em um teste secundário o elemento testado não é necessariamente um testador, pode ser qualquer dispositivo (*device*) ou serviço (*service*) na rede. Um teste secundário pode ser do tipo *device* ou *service*. Um serviço é qualquer processo executando em um testador, um dispositivo é qualquer recurso acessível da rede. É possível configurar um procedimento de testes específico para os serviços ou dispositivos que estão sendo testados.

A `Test-MIB` permite a especificação de um procedimento de testes projetado para cada entidade testada. O sistema assume que um procedimento de teste permite que um nodo sem-falhas determine corretamente o estado (*falho* ou *sem-falhas*) do elemento testado.

Procedimentos de testes específicos para os serviços e dispositivos testados podem ser definidos e utilizados. Quando um teste secundário é do tipo *device* e o seu testador falha, o algoritmo determina um outro testador para executar o teste do dispositivo. Se o teste secundário for do tipo *service* o algoritmo não atribui outro testador para assumir o teste secundário. Considera-se que um teste do tipo *service* somente pode ser realizado por um determinado testador. Neste caso o estado do serviço testado é considerado como desconhecido

(*unknown*).

De acordo com o algoritmo, os testes são executados em intervalos de testes, a `Test-MIB` permite a configuração deste intervalo para cada caso. Cada MIB também mantém informação sobre o número de intervalos de testes e o número de testes executados pelo testador. A MIB também mantém estatísticas que podem ser obtidas da monitoração distribuída do sistema, tais como o tempo médio que um nodo permanece sem-falha e o tempo médio necessário para um nodo ser reparado.

3.2.1 Descrição da Test-MIB

A `Test-MIB` é estruturada em 4 grupos, como mostra a figura 3.3: `tstTester`, `tstResults`, `tstStatistics` e `tstFullSecTestsDef`. Cada testador mantém objetos de gerência referentes a si mesmo e referentes a todos os outros agentes. O resultado dos testes executados em todo o sistema, por todos os testadores, é mantido nos objetos de gerência do grupo `tstResults`. Estatísticas são mantidas no grupo `tstStatistics`. O grupo `tstFullSecTestsDef` mantém a definição de todos os testes secundários de todos os testadores; este grupo é necessário porque qualquer testador pode ter que executar os testes secundários que eram executados por outro testador que se tornou falho.

A organização do grupo `tstTester` é mostrado na figura 3.4. Nos obje-

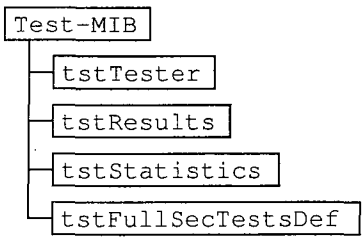

Figura 3.3: Test-MIB: os grupos de objetos.

tos deste grupo, cada testador mantém informações referentes a si mesmo.O objeto `tstOnOffSwitch` é um objeto de controle empregado para inicializar, reinicializar, parar ou continuar a operação do testador. Antes de um testador entrar em operação ele precisa ler arquivos de configuração pré-definidos.

O objeto `tstState` mantém o estado do testador: inicializando (*initializing*) ou sem-falhas (*fault-free*). No objeto `tstAddress` o testador mantém seu endereço IP (Internet Protocol). O objeto `tstInterval` é ajustado com número de segundos entre cada intervalo de testes, ou seja, o intervalo de tempo em que os testes são iniciados. O objeto `tstIntervalCounter` mantém o número de intervalos de testes que o testador já completou. Como um intervalo de testes pode consistir de um ou mais testes, o valor mantido pelo objeto `tstIntervalCounter` pode ser diferente do valor armazenado em `tstTestsCounter`, que armazena o número de testes executados. A contagem do objeto `tstTestsCounter` não inclui os testes secundários executados.

Abaixo um exemplo de uso dos objetos do grupo `tstTester` de um tes-

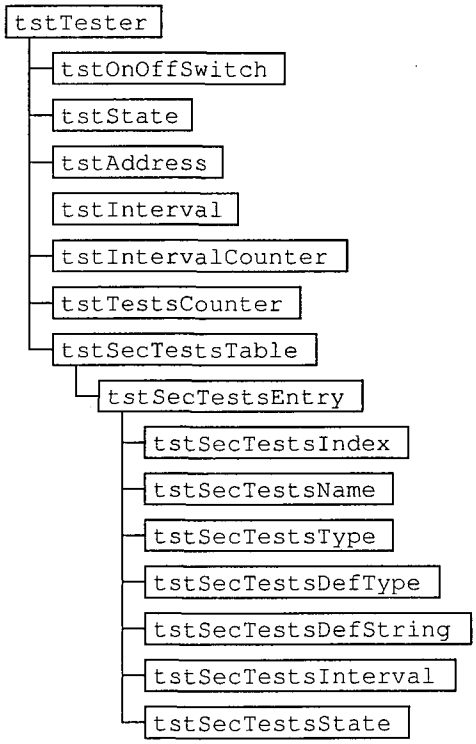

Figura 3.4: Test-MIB: o grupo tstTester.

tador. No exemplo o testador está executando o algoritmo, seu estado é sem-falhas, seu endereço IP é 200.17.212.103 e realizou 12 testes em 10 intervalos de testes desde a sua inicialização.

```
tstTester.tstOnOffSwitch.0 = on(0)
tstTester.tstState.0 = fault-free(0)
tstTester.tstAddress.0 = IpAddress: 200.17.212.103
tstTester.tstInterval.0 = Timeticks: (1000) 0:00:10.00
tstTester.tstIntervalCounter.0 = Counter32: 10
tstTester.tstTestsCounter.0 = Counter32: 12
```

O grupo `tstTester` também inclui `tstSecTestsTable`, que é uma tabela onde os testes secundários executados pelo testador são especificados. Esta informação é obtida tanto de um arquivo de configuração, como de ou-

tros testadores falhos de acordo com o algoritmo de diagnóstico. O objeto `tstSecTestsName` contém uma string que identifica o procedimento de teste executado. Um teste secundário pode ser de dois tipos: tipo *device* e tipo *service*; esta informação é guardada no objeto `tstSecTestsType`.

Dois objetos são empregados em conjunto para definir o procedimento de testes: `tstTestsDefType` e `tstTestsDefString`. Objeto `tstTestsDefType` é usado para determinar a interpretação do objeto `tstTestsDefString` e pode assumir o valor `execute` ou o valor `snmpget`. Se o valor for `execute` assume-se que o objeto `tstTestsDefString` contém uma string contendo um comando para ser executado pelo sistema operacional. Se o valor for `snmpget` assume-se que a string contida no objeto `tstTestsDefString` é a especificação de um *snmpget* e uma condição lógica aplicada ao resultado do *snmpget*. A string contém: o endereço IP de um agente, número da porta do agente, um identificador de objeto, seu tipo, e uma condição lógica para ser verificada durante o teste.

Abaixo um exemplo de uso da da tabela `tstSecTestsTable` contendo duas entradas que definem os testes secundário `"Printer Test"` e `"Switch Test"`. O teste `"Printer Test"` é definido como a execução do comando `printer_test.sh` a cada 10 segundos. O `"Switch Test"` é definido como uma mensagem *snmpget* requisitando o objeto `csPortsStatus` da MIB do agente com IP 200.17.212.3 e a aplicação da condição lógica `"==0"` ao valor

do objeto requisitado, em outras palavras, este teste é considerado sem-falhas somente se o valor do objeto `csPortsStatus` da MIB do agente com IP 200.17.212.3 for igual a 0.

```
tstSecTestsTable.tstSecTestsEntry.tstSecTestsIndex.1 = 1
tstSecTestsTable.tstSecTestsEntry.tstSecTestsIndex.2 = 2
tstSecTestsTable.tstSecTestsEntry.tstSecTestsName.1 = "Printer Test"
tstSecTestsTable.tstSecTestsEntry.tstSecTestsName.2 = "Switch Test"
tstSecTestsTable.tstSecTestsEntry.tstSecTestsType.1 = device(0)
tstSecTestsTable.tstSecTestsEntry.tstSecTestsType.2 = device(0)
tstSecTestsTable.tstSecTestsEntry.tstSecTestsDefType.1 = execute(0)
tstSecTestsTable.tstSecTestsEntry.tstSecTestsDefType.2 = snmpget(1)
tstSecTestsTable.tstSecTestsEntry.tstSecTestsDefString.1 = printer_test.sh
tstSecTestsTable.tstSecTestsEntry.tstSecTestsDefString.2 = csPortsStatus 151 public
200.17.212.3 i == 0
tstSecTestsTable.tstSecTestsEntry.tstSecTestsInterval.1 = Timeticks: (1000) 0:00:10.00
tstSecTestsTable.tstSecTestsEntry.tstSecTestsInterval.2 = Timeticks: (6000) 0:00:60.00
tstSecTestsTable.tstSecTestsEntry.tstSecTestsState.1 = faultfree(0)
tstSecTestsTable.tstSecTestsEntry.tstSecTestsState.2 = faultfree(0)
```

O intervalo entre duas execuções de um teste secundário é dado pelo objeto `tstSecTestsInterval`. O objeto `tstSecTestsState` contém o estado da entidade testada, que pôde ser falha (*faulty*), sem-falha (*fault-free*), desconhecido (*unknown*) ou erro de teste (*testError*). Se o testador ainda não pode obter informações da entidade testada, seja porque o teste não foi executado ou porque o teste é do tipo *service* de um testador falho, o estado é *unknown*. Se o procedimento de testes definido não pode ser executado o estado do teste é *testError*.

O grupo `tstResult` contém informação de diagnóstico sobre todos os testadores, dispostivo e serviços. A estrutura deste grupo é mostrada na figura 3.5. A tabela `tstReDiagsTable` guarda o endereço, estado e *timestamp* de todos os testadores. O *timestamp* é aquele definido pelo algoritmo *Hi-*

ADSD with Timestamps. O objeto `tstReDiagsIndex` que é índice da tabela é especialmente importante porque se refere aos identificadores seqüenciais dados a cada testador. A tabela `tstReSecTestsTable` guarda o resultado dos testes secundários.

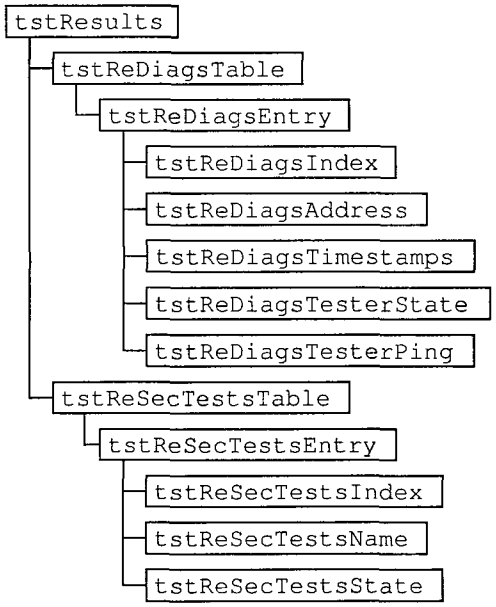

Figura 3.5: Test-MIB: o grupo tstResults.

A tabela `tstReDiagsTable` contém o objeto `tstReDiagsTesterState`, que reflete o estado do testador: *faulty*, *fault-free* ou *initializing.* Quando o estado de um testador é *falho*, um teste com ICMP (*Internet Control Message Protocol*) baseado em *echo requests* é executado. O resultado deste teste extra é mantido no objeto `tstReDiagsTesterPing`.

Um exemplo de uso da tabela `tstReDiagsTable` é dado abaixo. Cada entrada da tabela corresponde às informações de diagnóstico de um nodo ou

testador. O estado do testador 1, endereço IP 200.17.212.103, é sem-falha e o *timestamp* da informação de diagnóstico é igual 0. O estado do testador 3, endereço IP 200.17.212.104, é sem-falha e o *timestamp* é igual 4. O estado do testador 3, endereço IP 200.17.212.101, é falho, o resultado do teste de ICMP também é falho e o timestamp é 1.

```
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsIndex.1 = 1
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsIndex.2 = 2
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsIndex.3 = 3
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsAddress.1 = IpAddress: 200.17.212.103
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsAddress.2 = IpAddress: 200.17.212.104
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsAddress.3 = IpAddress: 200.17.212.101
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsTimestamps.1 = Counter32: 0
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsTimestamps.2 = Counter32: 1
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsTimestamps.3 = Counter32: 4
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsTesterState.1 = faultfree(0)
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsTesterState.2 = faultfree(0)
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsTesterState.3 = fauly(0)
tstReDiagsTable.tstRedDiagsEntry.tstReDiagsTesterPing.3 = faulty(1)
```

A tabela `tstReSecTestsTable` consiste de objetos que mantêm informações sobre todos os testes secundários executados no sistema. Esta tabela é composta por dois objetos: `tstReSecTestsName` e `tstReSecTestsState`. O objeto `tstReSecTestsName` contém uma string que identifica o procedimento de teste executado. O estado da entidade testada é guardada no objeto `tstReSecTestsState`, que pode assumir o valor *faulty*, *fault-free*, *unknown* ou *testError*. Se o testador ainda não pôde obter informações do componente testado, seja porque o teste não foi executado, seja porque o teste é do tipo *service* de um testador falho, o estado é *unknown*. Se o procedimento de testes resultou em erro, o estado é *testError*.

Um exemplo de uso da tabela `tstReSecTestsTable` é dado abaixo. No

exemplo são mostrados os resultados e o nome de quatro testes secundários. Os testes "Printer Test" e "Switch Test" são executados pelo testador 1. Os testes "DNS Test" e "(3.1) NFS Tests" são executado pelo testador 2. A *string* "(3.1)" no objeto tstReSecTestsName.2.2 era inicialmente o teste secundário índice 1 do testador 3.

```
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsIndex.1.1 = 1
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsIndex.1.2 = 2
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsIndex.2.1 = 1
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsIndex.2.2 = 2
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsName.1.1 = "Printer Test"
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsName.1.2 = "Switch Test"
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsName.2.1 = "DNS Test"
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsName.2.2 = "(3.1) NFS Test"
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsState.1.1 = faultfree(0)
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsState.1.2 = faultfree(0)
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsState.2.1 = faultfree(0)
tstReSecTestsTable.tstReSecTestsEntry.tstReSecTestsState.2.2 = faultfree(0)
```

O grupo tstStatistics mantém as estatísticas do sistema calculadas a partir dos resultados do diagnóstico. A estrutura do grupo é mostrada na figura 3.6. A tabela tstStatsTable mantém estatísticas geradas da informação contida na tabela tstReDiagsTable. Objeto tstStatsMTFaulty mantém o tempo médio observado no qual um nodo se mantém falho e tstStatsMTFFree mantém o tempo médio observado no qual um nodo permanece sem-falhas.

```
tstStatsTable.tstStatsEntry.tstStatsMTFaulty.1 = Timeticks: (530103) 01:28:01.03
tstStatsTable.tstStatsEntry.tstStatsMTFaulty.2 = Timeticks: (1430631) 03:58:06.31
tstStatsTable.tstStatsEntry.tstStatsMTFaulty.3 = Timeticks: (1234511) 03:25:45.11
tstStatsTable.tstStatsEntry.tstStatsMTFaulty.4 = Timeticks: (723415) 02:00:34.15
tstStatsTable.tstStatsEntry.tstStatsMTFFree.1 = Timeticks: (68478929) 7 days, 22:13:09.29
tstStatsTable.tstStatsEntry.tstStatsMTFFree.2 = Timeticks: (67578401) 7 days, 19:43:04:01
tstStatsTable.tstStatsEntry.tstStatsMTFFree.3 = Timeticks: (67774521) 7 days, 20:15:45.21
tstStatsTable.tstStatsEntry.tstStatsMTFFree.4 = Timeticks: (68285617) 7 days, 21:40:56:17
```

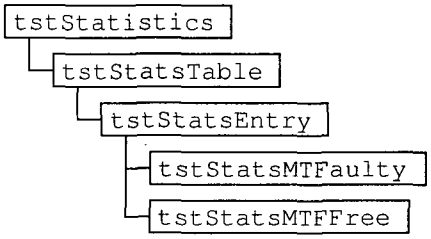

Figura 3.6: Test-MIB: o grupo tstStatistics.

O grupo `tstFullSecTestsDef` mantém a tabela `tstFSTDTable` que contém a definição de todos testes secundários executados no sistema. A estrutura geral deste grupo pode ser vista na figura 3.7. Estes objetos são necessários porque um teste secundário de um testador falho pode ser atribuído para um outro testador sem-falhas.

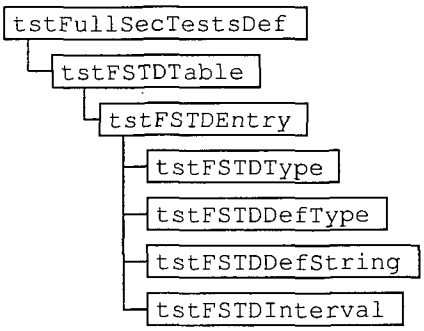

Figura 3.7: Test-MIB: o grupo tstFullSecTestsDef.

Abaixo é mostrado um exemplo de uso da tabela `tstFSTDTable` contendo informações de definição de todos testes secundários executados em um sistema.

```
tstFSTDTable.tstFSTDEntry.tstFSTDType.1.1 = device(0)
tstFSTDTable.tstFSTDEntry.tstFSTDType.1.2 = device(0)
tstFSTDTable.tstFSTDEntry.tstFSTDType.2.1 = device(0)
```

```
tstFSTDTable.tstFSTDEntry.tstFSTDType.3.1 = device(0)
tstFSTDTable.tstFSTDEntry.tstFSTDDefType.1.1 = execute(0)
tstFSTDTable.tstFSTDEntry.tstFSTDDefType.1.2 = snmpget(1)
tstFSTDTable.tstFSTDEntry.tstFSTDDefType.2.1 = execute(0)
tstFSTDTable.tstFSTDEntry.tstFSTDDefType.3.1 = execute(0)
tstFSTDTable.tstFSTDEntry.tstFSTDDefString.1.1 = printer_test.sh
tstFSTDTable.tstFSTDEntry.tstFSTDDefString.1.2 = csPortsStatus 151 public
200.17.212.3 i == 0
tstFSTDTable.tstFSTDEntry.tstFSTDDefString.2.1 = dns_test.sh
tstFSTDTable.tstFSTDEntry.tstFSTDDefString.3.1 = nfs_test.sh
tstFSTDTable.tstFSTDEntry.tstFSTDInterval.1.1 = Timeticks: (1000) 0:00:10.00
tstFSTDTable.tstFSTDEntry.tstFSTDInterval.1.2 = Timeticks: (6000) 0:00:60.00
tstFSTDTable.tstFSTDEntry.tstFSTDInterval.2.1 = Timeticks: (12000) 0:02:00.00
tstFSTDTable.tstFSTDEntry.tstFSTDInterval.1.2 = Timeticks: (3000) 0:00:3.00
```

A descrição completa em ASN.1 da `Test-MIB` pode ser encontrada no apêndice A.

Capítulo 4

Uma Ferramenta Confiável para Gerência de Redes Locais

Uma ferramenta para gerência de redes locais baseada em diagnóstico distribuído foi desenvolvida e é descrita neste capítulo. A ferramenta desenvolvida é baseada em SNMP e implementa o algoritmo *Hi-ADSD with Timestamps*. A ferramenta pode ser utilizada para monitorar redes locais e construir aplicações tolerantes a falhas baseadas no sistema de diagnóstico. Inicialmente é apresentada a arquitetura da ferramentada desenvolvida. Em seguida, são apresentados resultados da utilização da ferramenta na monitoração de uma rede local. Finalmente é descrita a construção de um servidor Web tolerante a falhas baseado na ferramenta desenvolvida.

4.1 A Ferramenta Desenvolvida

A ferramenta de diagnóstico distribuído desenvolvida implementa o algoritmo *Hi-ADSD with timestamps* utilizando o protocolo SNMP. A ferramenta foi codificada utilizando linguagem C e o pacote de domínio público NET-SNMP [23]. Um *testador* é uma unidade da ferramenta que executa o algoritmo de diagnóstico distribuído.

Acessando as informações disponibilizadas pela ferramenta o gerente humano pode determinar quais elementos monitorados estão *falhos* e quais estão *sem-falhas*. A ferramenta pode ser acessada através de uma interface Web. Esta interface Web obtém informações de diagnóstico, periodicamente, a partir de qualquer um dos agentes que executa a ferramenta.

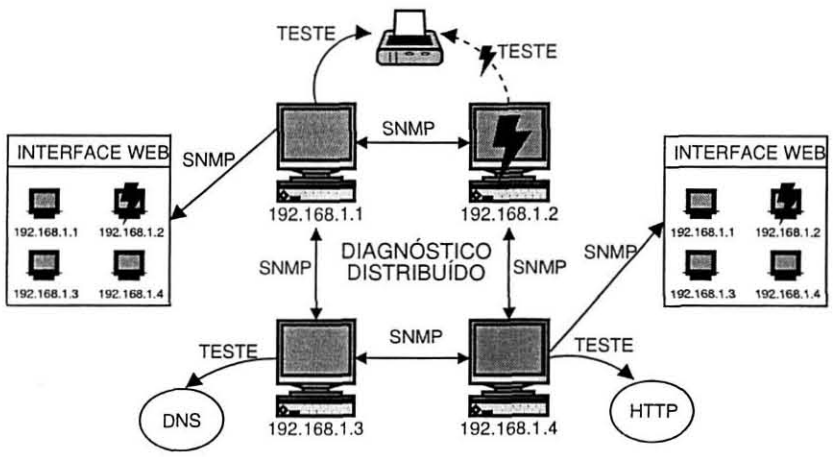

Figura 4.1: Utilização da ferramenta de diagnóstico distribuída.

A figura 4.1 mostra em um exemplo a arquitetura da ferramenta desenvolvida. O sistema é composto por 4 testadores que executam o algoritmo de diagnóstico distribuído e realizam testes entre si e o servidor de DNS, o servidor de HTTP e a impressora. Na figura o testador original da impressora está falho e deixou de testá-la, neste caso a ferramenta atribuiu a tarefa de testar a impressora para outro testador.

Cada testador é um agente SNMP que executa o algoritmo *Hi-ADSD with timestamps* e corresponde a um nodo do algoritmo. A figura 4.2 mostra a estrutura de um testador. As estruturas de dados utilizadas pelo *Hi-ADSD* são disponibilizadas pela `Test-MIB` descrita no capítulo anterior. Um testador testa e obtém informações de diagnóstico de outro testador acessando a `Test-MIB` utilizando o protocolo SNMP.

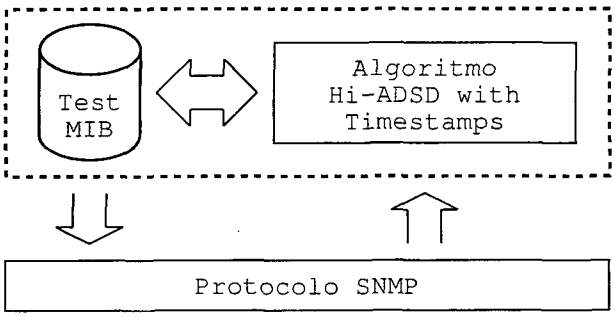

Figura 4.2: Estrutura de um testador.

O algoritmo *Hi-ADSD with Timestamps* determina os testes realizados por cada testador. Este testes são chamados *testes primários.* Um teste é

feito utilizando uma mensagem `snmpget` para unidade que se deseja testar. Se a unidade responder corretamente à mensagem, seu estado é considerado sem-falhas (*fault-free*), caso contrário o estado é considerado falho (*faulty*). Se o teste utilizando SNMP falha então é realizado um teste utilizando ICMP.

Quando um testador realiza um teste primário e o nodo testado está sem-falha são obtidas as informações de diagnóstico previstas pelo algoritmo *Hi-ADSD with Timestamps*. As informações são obtidas acessando a `Test-MIB` no nodo testado através de uma requisição `snmpgetbulk`. As informações obtidas são tratadas pelo *Hi-ADSD with Timestamps* que atualiza a `Test-MIB` do testador.

Os testes primários permitem determinar o estado de todos testadores do sistema. Além destes testes é possível definir *testes secundários*. Os testes secundário permitem testar componentes que não são capazes de executar o algoritmo, mas devem ser monitorados. Estes testes são definidos na `Test-MIB`. Quando um testador falha um outro testador assume a execução do teste secundário. Assim é possível garantir a execução de todos testes secundários mesmo quando apenas um testador estiver sem-falhas.

A ferramenta tem sido utilizada por vários meses para monitorar o funcionamento de parte do laboratório do Departamento de Informática da Universidade Federal do Paraná. Os laboratórios monitorados contavam com 67 máquinas baseadas em diferentes processadores Intel Pentium, AMD K6 e

Intel 486, interligados por uma rede Ethernet de 100Mpbs.

Para monitorar os laboratórios descritos foi utilizado um intervalo de testes de 10 segundos. O impacto na performance dos processadores executando o agente foi medido. O consumo de CPU esteve sempre abaixo de 3%, mesmo nas maquinas que utilizavam processadores Intel 486. A largura de banda utilizada ficou abaixo de 0.001% da banda total.

4.2 Monitoramento de Rede

A seguir são descritos os resultados práticos obtidos através da monitoração distribuída dos laboratórios descritos acima durante vários meses. Um número de eventos artificiais foi injetado com objetivo de validar a implementação. Estes experimentos também permitiram medir a latência em segundos, que é um resultado mais intuitivo do que as medidas teóricas dadas em intervalos de testes.

Dois conjuntos de experimentos são descritos. O primeiro conjunto é composto por 500 experimentos, onde *um* nodo escolhido de forma aleatória se torna falho e depois se recupera. O segundo conjunto foi composto de 700 experimentos nos quais *um número aleatório* de nodos sem-falhas se tornam falhos e depois se recuperam. Em ambos conjuntos de experimentos a latência para todos os nodos sem-falhas detectarem os eventos de falha foi

Evento	*Latência Mínima*	*Latência Média*	*Latência Máxima*
falha	59	124,5	345
recuperação	69	121	297

Tabela 4.1: Latência em segundos para diagnosticar um evento em um nodo.

calculada, assim como a latência para todos nodos sem-falhas detectarem os eventos de recuperação.

A latência mínima, máxima e média observada no primeiro conjunto de experimentos é mostrada na tabela 4.1. A primeira linha mostra a latência para um evento de falha. A latência mínima observada foi de 59 segundos para todos nodos sem-falhas diagnosticarem a falha de um nodo sem-falha. A latência máxima foi 345 segundos, enquanto a latência média foi 124.5 segundos. A segunda linha mostra a latência de diagnóstico para o evento de recuperação. A latência mínima observada foi 69 segundos e a latência máxima observada foi 297 segundos, enquanto a latência observada média foi 121 segundos.

O gráfico na figura 4.3 mostra a função de distribuição cumulativa da latência observada no primeiro conjunto de experimentos. Mais de 90% dos experimentos realizados com o evento de falha foram diagnosticados em até 150 segundos. Mais de 90% dos experimentos com o evento de recuperação foram diagnósticado em até 158 segundos.

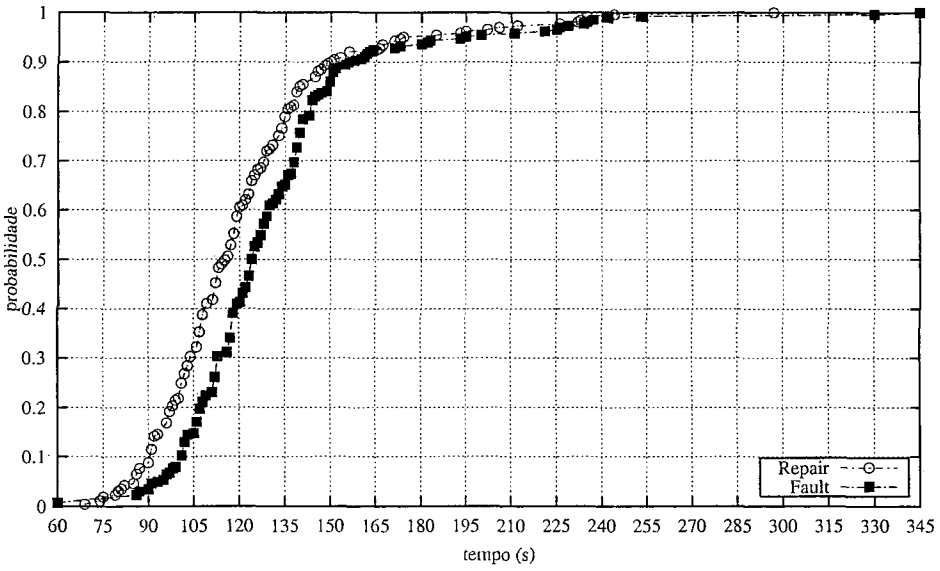

Figura 4.3: Função de distribuição cumulativa da latência para diagnosticar um evento em um nodo.

O segundo conjunto de experimentos mediu a latência de diagnóstico de múltiplos eventos de falha e recuperação simultâneos. Tanto o número de nodos como a escolha dos nodos que falhavam a cada experimento foi determinada aleatoriamente. Estes nodos posteriormente se recuperavam. A tabela 4.2 mostra a latência mínima, média e máxima observada. A latência mínima observada foi 43 segundos e a latência máxima 338 segundos, a latência média ficou em 125 segundos.

O gráfico na figura 4.4 mostra a função de distribuição cumulativa da latência observada no segundo conjunto de experimentos. Mais de 90% dos experimentos realizados com o evento de falha foram diagnósticados em até

Evento	*Latência Mínima*	*Latência Média*	*Latência Máxima*
falha	43	182	381
recuperação	43	125	338

Tabela 4.2: Latência em segundos para diagnosticar eventos em um aleatório de nodos.

241 segundos. Mais de 90% dos experimentos com o evento de recuperação foram diagnósticado em até 148 segundos.

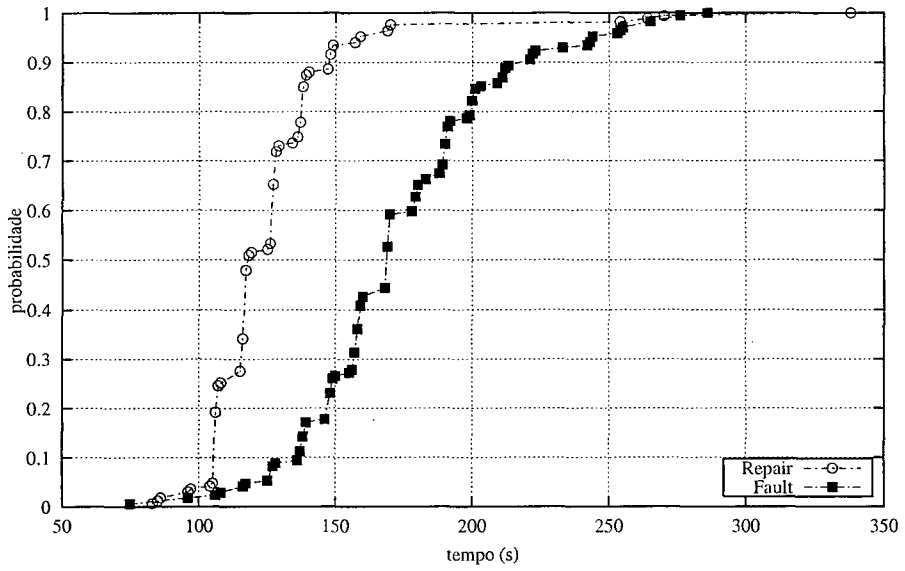

Figura 4.4: Função de distribuição cumulativa da latência para diagnosticar eventos em um número aleatório de nodos.

A latência de diagnóstico observada neste experimento esteve sempre abaixo do máximo teórico de 490 segundos. A latência média para todos nodos sem-falhas completarem o diagnóstico de um evento ficou em torno de

2 minutos. A latência média para todos os nodos sem-falha completarem o diagnóstico de múltiplos eventos simultâneos ficou por volta de 3 minutos. Estes resultados mostram que a ferramenta apresenta uma solução prática para monitoramento confiável de rede locais. Naturalmente é possível melhorar este quadro reduzindo o intervalo de teste, que neste experimento foi de 10 segundos, tendo um impacto muito baixo no desempenho do sistema.

4.3 Um Servidor HTTP Tolerante a Falhas

A ferramenta de monitoramento confiável pode ser utilizada como suporte para implementar aplicações de rede tolerantes a falhas. Um experimento com o servidor HTTP (*HyperText Transfer Protocol*) Apache é descrito. A escolha deste protocolo se deve ao fato da aplicação Web ter se tornado imprescindível para organizações motivando a utilização de sistemas para garantir sua disponibilidade.

Um grupo de máquinas é definido, neste grupo cada membro é potencialmente capaz de executar um servidor Apache. Inicialmente o Apache executa em uma determinada máquina deste grupo. Esta máquina recebe um endereço IP virtual além de seu próprio endereço IP. O endereço IP virtual "segue" o servidor Apache, sendo atribuído para o membro do grupo que estiver executando o servidor. O endereço virtual é atribuído utilizando

a técnica de IP *aliasing* do sistema operacional Linux, que permite atribuir mais de um endereço IP para uma interface de rede. A figura 4.5 mostra um grupo de quatro máquinas implementado um servidor Web tolerante a falhas.

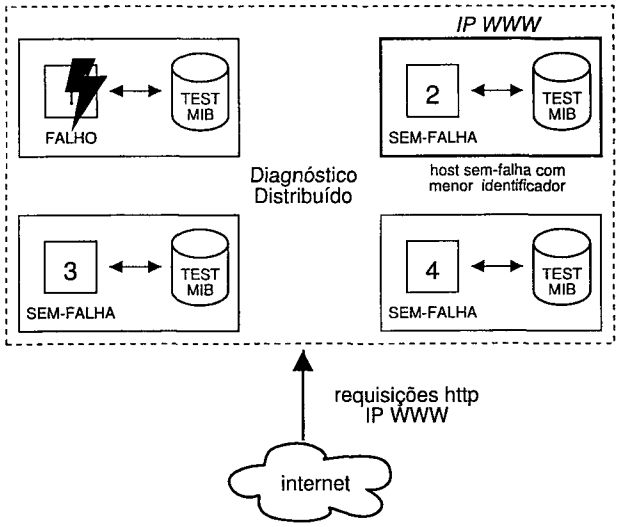

Figura 4.5: Um grupo de quatro máquinas implementa o servidor Web tolerante a falhas.

Todas as máquinas do grupo utilizam a Test-MIB para se monitorarem. Um script em cada máquina monitora a Test-MIB procurando pelo membro sem-falhas do grupo com o menor identificador. Quando uma máquina descobre que este identificador corresponde ao seu próprio identificador então a máquina atribui para si o IP virtual que "segue" o Apache, então inicializa o servidor Apache e passa a atender as requisições HTTP.

Existe a necessidade de que os membros do grupo mantenham os arqui-

vos acessados pelo servidor Web atualizados. Neste experimento a ferramenta *rsync* é utilizada para sincronizar o conteúdo dos servidor Web. O rsync utiliza um protocolo de atualização remoto que permite que somente as diferenças entre os arquivos fontes e destino sejam transferidos pela rede.

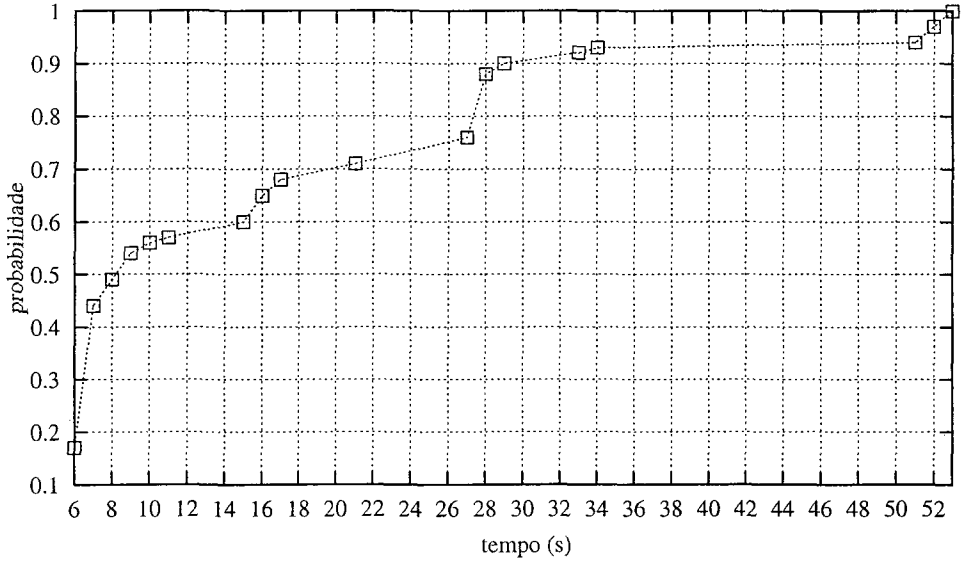

Figura 4.6: A função de distribuição cumulativa do tempo de recuperação do servidor Web.

O experimento foi executado em seis máquinas e foram realizadas medições por um período de 12 horas e 23 minutos. Cada um dos 6 membros foi derrubado aleatoriamente, a probabilidade de cada máquina estar disponível em um dado momento variou de 13% até 66%

No período observado o servido Apache foi derrubado 74 vezes. O gráfico na figura 4.6 mostra a função de distribuição cumulativa do tempo de re-

cuperação do servidor, ou seja, o intervalo de tempo transcorrido entre a queda do servidor apache até que uma outra máquina do grupo detectar a falha do servidor e começar a atender requisições HTTP. A despeito do número de falhas, o serviço Web se manteve disponível durante 97.35% do tempo; o intervalo médio de tempo em que o serviço estava indisponível foi 16 segundos.

Capítulo 5

Conclusão

Sistemas de gerência de rede permitem o controle e monitoramento de redes. Este trabalho apresentou uma ferramenta de gerência confiável e distribuída baseada em diagnóstico distribuído implementada com o protocolo SNMP. O algoritmo de diagnóstico distribuído utilizado foi o algoritmo *Hi-ADSD with Timestamps*. Entre as contribuições deste trabalho está a definição de uma estratégia de testes no algoritmo *Hi-ADSD with Timestamps* que permite a monitoração confiável de unidades que não executam o algoritmo. A ferramenta utiliza agentes SNMP que são capazes de testar componentes da rede e trocar informações de diagnóstico de acordo com o algoritmo de diagnóstico distribuído.

Uma MIB para o sistema distribuído de testes também foi proposta e implementada pela ferramenta. A MIB permite especificar procedimentos

de testes projetados para o componente testado. As estruturas de dados necessárias para o algoritmo de diagnóstico também são previstas na MIB.

A ferramenta desenvolvida tem sido utilizada na prática para monitorar parte dos laboratórios do Departamento de Informática da Universidade Federal do Paraná já durante vários meses. No período observado foram monitoradas 67 máquinas rodando o sistema operacional Linux. O impacto na performance dos processadores executando o agente foi medido. O consumo de CPU esteve sempre abaixo de 3%, mesmo em máquinas que utilizavam processadores Intel 486. A largura de banda de rede utilizada ficou abaixo de 0.001% da banda total.

Experimentos também permitiram medir o tempo necessário para que a ferramenta diagnostica-se eventos de falhas e recuperação. Dado o intervalo de testes de 10 segundos a latência para diagnosticar um evento foi 123 segundos. A latência para diagnosticar múltiplos eventos simultâneos foi 154 segundos. Estes resultados mostraram que a ferramenta é uma solução extremamente viável para monitoramento confiável de redes locais.

Como exemplo da utilização da ferramenta como suporte para a construção de aplicações tolerantes a falhas um experimento foi feito com o servidor Web Apache. No experimento o servidor Apache foi derrubado 74 vezes, a despeito destas falhas, o Apache esteve disponível durante 97.35% do tempo; o intervalo médio de tempo em que o serviço ficou indisponível foi

16 segundos.

Trabalho futuros incluem a criação de um framework que permita a construção de aplicações distribuídas tolerantes a falhas baseadas em diagnóstico distribuído.

Referências Bibliográficas

[1] D. Harrington, R. Presuhn, and B. Wijnen, "An Architecture for Describing SNMP Management Frameworks," *Request for Comments 2271*, January, 1998.

[2] A. Leinwand, and K. Fang-Conroy, *Network Management: A Practical Perspective,* Addison-Wesley, Reading, MA, 1995.

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

[4] E.P. Duarte Jr. and A.L. dos Santos, "Semi-Active Replication of SNMP Objects in Agent Groups Applied for Fault Management," *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management (IM'01)*, Seattle WA, 2001

[5] *Distributed Management (DisMan) Charter*, http://www.ietf.org/html.charters/disman-charter.html

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

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

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

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

[10] E.P. Duarte Jr., L.C.P. Albini, and A. Brawerman, "An Algorithm for Distributed Diagnosis of Dynamic Fault and Repair Events," *In Proceedings of the 7th IEEE International Conference on Parallel and Distributed Systems, IEEE/ICPADS'00*, pp. 299-306, Iwate, Japan, 2000.

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

[12] E.P. Duarte Jr., F. Mansfield, T Nanya, and S Noguchi, "Non-Broadcast Network Fault-Monitoring Based on System-Level Diagnosis," *Proc. IFIP/IEEE International Symposium on Integrated Network Management (IM'97)*, pp. 597-609, 1997.

[13] R.P. Bianchini, and R. Buskens, "An Adaptive Distributed System-Level Diagnosis Algorithm and Its Implementation," *Proc. FTCS-21*, pp. 222-229, 1991.

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

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

[16] E.P. Duarte Jr., A. Brawerman, and L.C.P. ALbini, "A Diagnosis Algorithm based on Clusters with Detours," *Available at http://www.inf.ufpr.br/ elias.*

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

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

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

[20] W. Stallings, *Snmp, Snmpv2, Snmpv3 and Rmon 1 and 2,* Addison-Wesley, Reading, MA, 1999.

[21] A. S. Tanenbaum, *Computer Networks,* P T R Prentice Hall, 1996.

[22] *The Apache Software Foundation,* http://www.apache.org.

[23] *The NET-SNMP Home Page,* http://net-snmp.sourceforge.net.

[24] *The RSYNC Home Page,* http://samba.anu.edu.au/rsync.

Apêndice A

Descrição ASN.1 da Test-MIB

```
TEST-MIB DEFINITIONS ::= BEGIN

IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE,
    Unsigned32
        FROM SNMPv2-SMI

    RowStatus
        FROM SNMPv2-TC

    MODULE-COMPLIANCE, OBJECT-GROUP
        FROM SNMPv2-CONF

    SnmpAdminString
        FROM SNMP-FRAMEWORK-MIB;

testMIB MODULE-IDENTITY
    LAST-UPDATED "200010120000Z"
    ORGANIZATION "Federal University of Parana' - Dept. Informatics"
    CONTACT-INFO
        "Luis Carlos Erpen de Bona
         Elias P. Duarte Jr.
         Federal University of Parana'
         Dept. Informatics
         P.O. Box 19018
         Curitiba, PR 81531-990
         Brazil
         Phone: +55-41-267-5244
         Email: {bona, elias}@inf.ufpr.br"
     DESCRIPTION
       " This MIB module defines of a set of objects that supports the
       specification of distributed tests, as well as the data structures
       required to perform distributed hierarchical adaptive diagnosis. "

    ::= { yyy     xxx }              -- to be assigned by IANA
```

```
--
-- This Node group information
--
tstTester               OBJECT IDENTIFIER ::= { testMIB 1 }
tstResults              OBJECT IDENTIFIER ::= { testMIB 2 }
tstStatistics           OBJECT IDENTIFIER ::= { testMIB 3 }
tstFullSecTests         OBJECT IDENTIFIER ::= { testMIB 4 }

tstOnOffSwitch OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
      "This Object is a control object that is employed to initialize,
       reset, stop or resume the tester operation. When a tester starts
       operating it read pre-defined configuration files"
    ::= {tstTester 1}

tstState OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      " This object indicates whether this node is fault-free (0)
        or initializing (2).  In this context, (1) stands for faulty,
        and is not assumed to be kept by the managed object."
    ::= {tstTester 2}

tstAddress OBJECT-TYPE
    SYNTAX      IpAddress
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
        " The IPAddress agent of this node"
    ::= {tstTester 3}

tstInterval OBJECT-TYPE
    SYNTAX      Counter32
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
      " This objects keeps the testing interval, i.e., the interval of time
        in which a testing round is executed by this node"
    ::= {tstTester 4}

tsIntervalCounter OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      " Number of testing intervals executed by this node since
        its initialization."
    ::= {tstTester 5}

tstTestsCounter OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      " Number of tests executed by this node since its
```

```
        initialization. Its value may differ from de value
        kept in tstTestsCounter since a testing interval
        may consist of one or more tests. This counter does
        not include the seconday tests"
    ::= {tstTester 6}


--
-- this node Test Table definition
--

tstSecTestsTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF TsTestsEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "This tables specifies the secondary tests executed by this tester.
        The table is filled with information obtained both from a
        configuration file, and from other faulty testers according to the
        diagnosis algorithm."
    ::= {tstTester 7}

tstSecTestsEntry OBJECT-TYPE
    SYNTAX      TstSecTestsEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "Each entry contains the specification of one secondary test executed
         by this tester"
    INDEX { tsTestsIndex }
    ::= { tsTestsTable 1 }

TstSecTestsEntry ::= SEQUENCE
{
    tstSecTestsIndex     Unsigned32,
    tstSecTestsName      SnmpAdminString,
    tstSecTestsType      Unsigned32,
    tstSecTestsDeftype   SnmpAdminString,
    tstSecTestsDefstring SnmpAdminString,
    tstSecTestsInterval  Unsigned32,
    tstTestsState        Unsigned32
}

tstSecTestsIndex OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
    STATUS     current
    DESCRIPTION
      "A unique identifier for each secondary test"
    ::= { tstSecTestsEntry 1}

tstSecTestsName OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
    STATUS     current
    DESCRIPTION
      "A string that identifies the secondary test"
    ::= { tstSecTestsEntry 2}

tstSecTestsType OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
```

```
    STATUS      current
    DESCRIPTION
         "There is two type: service(1) or device(2). If the tester
        becomes faulty(1) and the secondary test type is device(2)
        another tester will be assigned to continue this tests execution.
        In other case, the test type is device(2) the test execution
        is stopped"
    ::= { tstSecTestsEntry 3}

tstSecTestsDeftype OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS read-only
    STATUS      current
    DESCRIPTION
      "This object gives the test procedure type. A test could be of
       two type: execute or snmpget. The test procedure type determine
       how the object tstSecTestsDefstring is interpreted"
    ::= { tstSecTestsEntry 4}

tstSecTestsDefstring OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS read-only
    STATUS      current
    DESCRIPTION
      "This object meaning changes according with the test procedure
       type kept in object tstSecTestsDeftype. If the value is execute,
       this object assumed to keep a shell command to be executed by
       the operating system.  If the value is snmpget the string in
       object tstTestsDefString is assumed to contain both an SNMP agent
       address, port number, and the object identifier, its type, and a
       logical condition to be checked during the test.

       Sintaxe:
         SNMPG AGENT PORT COMMUNITY ObjectID TYPE LOGICAL_OPERATOR VALUE

       The type is a single character, one of:
              i  INTEGER
              u  UNSIGNED
              s  STRING
              x  HEX STRING
              d  DECIMAL STRING
              n  NULLOBJ
              o  OBJID
              t  TIMETICKS
              a IPADDRESS "
    ::= { tstSecTestsEntry 5}

tstSecTestsInterval OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS read-only
    STATUS      current
    DESCRIPTION
         "The time interval beetwen two test executions"
    ::= { tstSecTestsEntry 6}

tstSecTestsState OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS read-only
    STATUS      current
    DESCRIPTION
       "The last execution test result"
    ::= { tstSecTestsEntry 7}
```

```
--
--  Results Group
--

tstReDiagsTable OBJECT-TYPE
    SYNTAX       SEQUENCE OF TsRsDiagEntry
    MAX-ACCESS   not-accessible
    STATUS       current
    DESCRIPTION
        "This table keeps all testers addresses, states, and timestamps.
         The index of this table is especially important as it refers to
         the sequential identifier given to each tester"
    ::= {tstResults 1}

tstReDiagsEntry OBJECT-TYPE
    SYNTAX      TstReDiagEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "Each entry contains identification and diagnosisis informations
         about a the tester"
    INDEX { tstReDiagsIndex }
    ::= { tstReDiagsTable 1 }

TstReDiagEntry ::= SEQUENCE
{
    tstReDiagsIndex        Unsigned32,
    tstReDiagsAddress      IpAddress,
    tstReDiagsTimestamps   Unsigned32,
    tstReDiagsTesterState  Unsigned32,
    tstReDiagsTesterPing   Unsigned32
}

tstReDiagsIndex OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      "Each tester is given a sequential identifier"
    ::= {tstReDiagsEntry 1}

tstReDiagsAddress OBJECT-TYPE
    SYNTAX      IpAddress
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      "The tester agent IP address"
    ::= {tstReDiagsEntry 2}

tstReDiagsTimestamps OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      "The timestamps are those defined by the diagnosis algorithm
       HI-ADSD with timestamps"
    ::= {tstReDiagsEntry 3}

tstReDiagsTesterState OBJECT-TYPE
    SYNTAX      Counter32
    MAX-ACCESS  read-only
```

```
    STATUS      current
    DESCRIPTION
      "This objects kept the tester state: faulty, fault-free or  initializing"
    ::= {tstReDiagsEntry 4}

tstReDiagsTesterPing OBJECT-TYPE
    SYNTAX      IpAddress
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      "Whenever the state of a given tester is faulty,  an ICMP (Internet
       Control Message Protocol) test is executed, which is based
       on echo requests. This object keeps this extra test result"
    ::= {tstReDiagsEntry 5}


--
-- Tests Results Table
--

tstReSecTestsTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF TsRsTestsEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "This table keeps information about all secondary tests that are
        being executed in the system"
    ::= {tstResults 2}

tstReSecTestsEntry OBJECT-TYPE
    SYNTAX      TstReSecTestsEntry
    MAX-ACCESS   not-accessible
    STATUS      current
    DESCRIPTION
        "Each entry has information about a secondary test executed"
    INDEX { tstReDiagsIndex, tstReSecTestsIndex}
    ::= { tstReSecTestsTable 1 }

TstReSecTestsEntry ::= SEQUENCE
{
    tstReSecTestsIndex      Unsigned32,
    tstReSecTestsName       SnmpAdminString,
    tstReSecTestsState      Unsigned32
}

tstReSecTestsIndex OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
    STATUS     current
    DESCRIPTION
      "This table keeps information about the results
       of all secondary tests executed in the system"
    ::= { tstReSecTestsEntry 1}

tstReSecTestsName OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
    STATUS     current
    DESCRIPTION
      "The name of the secondary test"
    ::= { tstReSecTestsEntry 2}
```

```
tstReSecTestsState OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
    STATUS     current
    DESCRIPTION
      "The result of the last secondary test execution"
    ::= { tstReSecTestsEntry 3}

--
--  Statistcs Table
--

tstStatsTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF TsStatsEntry
    MAX-ACCESS   not-accessible
    STATUS      current
    DESCRIPTION
      " This table keeps system's statistics computed from
        diagnosis results"
    ::= {tstStatistics 1}

tstStatsEntry OBJECT-TYPE
    SYNTAX      TsStatsEntry
    MAX-ACCESS   not-accessible
    STATUS      current
    DESCRIPTION
        "Each entry contains statistical information about a node extracted
         of this node diagnose data"
    AUGMENTS { tstReDiagsEntry }
    ::= { tstStatsTable 1 }

TsStatsEntry ::= SEQUENCE
{
    tstStatsMTFaulty      TimeTicks,
    tstStatsMTFFree       TimeTicks
}

tstStatsMTFaulty OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      "Observed mean time a node remains in state faulty"
    ::= {tstStatsEntry 1}

tstStatsMTFFree OBJECT-TYPE
    SYNTAX      TimeTicks
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
      "Observed mean time a node remains in state fault-free"
    ::= {tstStatsEntry 2}

--
--
--  Exchange Table
--
--

tstFullSecTestsDef OBJECT-TYPE
    SYNTAX      SEQUENCE OF TstFullSecTestsDef
    MAX-ACCESS   not-accessible
```

```
    STATUS      current
    DESCRIPTION
        "This group keeps information about all secondary
         tests specification"
    ::= { tstFullSecTestsDef 1 }

--
-- Exchange Tests Table
--

tstFSTDTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF TstExFSTDEntry
    MAX-ACCESS   not-accessible
    STATUS      current
    DESCRIPTION
        "This tables has the definition informations about the set
        of tests executed by this agent."
    ::= {tstFullSecTestsDef 1}

tstFSTDEntry OBJECT-TYPE
    SYNTAX      TsExTestsEntry
    MAX-ACCESS   not-accessible
    STATUS      current
    DESCRIPTION
        "Each entry contains the definition of a test executed
        by an agent"
    AUGMENTS { tstReSecTestsEntry }
    ::= { tstFSTDTable 1 }

TstFSTDEntry ::= SEQUENCE
{
    tstFSTDType           Unsigned32,
    tstFSTDDeftype   SnmpAdminString,
    tstFSTDDefstring SnmpAdminString,
    tstFSTDInterval       Unsigned32
}

tstFSTDType OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
    STATUS     current
    DESCRIPTION
        "The secondary test type: service(1) or device(2)"
    ::= { tstFSTDEntry 1}

tstFSTDDeftype OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
    STATUS     current
    DESCRIPTION
        "The secondary test procedure type: execute(1) or snmpget(2)"
    ::= { tstFSTDEntry 2}

tstFSTDDefstring OBJECT-TYPE
    SYNTAX     Unsigned32
    MAX-ACCESS read-only
    STATUS     current
    DESCRIPTION
        "The procedure definition"
    ::= { tstFSTDEntry 3}
```

```
tstFSTDInterval OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS read-only
    STATUS      current
    DESCRIPTION
        "The time interval beetwen two test executions"
    ::= { tstFSTDEntry 4}
END
```

Apêndice B

Código Fonte C da Ferramenta

```
/* some important SNMP headers */
#include <config.h>
#include "../mibincl.h"
#include "../struct.h"
#include "../../../snmplib/system.h"
#include "util_funcs.h"

#include "readconfigs.h"
#include "list.h"
#include "bulk.h"
#include "timestamps.h"

#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/time.h>

#include <pthread.h>

#define NODATA   0
#define UPDATED  1
#define UPDATING 2


int iptoint(char* _ip, char* error);
void GET_INFO(nodo* tester, int node,
              char* TESTED_State, char* TESTED_DIAGNOSE_LIST,
              unsigned int * TESTED_TIMESTAMP_LIST);
int ping(char *host);
int execute(char *program);
void removeextratests(int node);
int  getthisnodeid ();
void freectval(mibtests* auxct);
void START_CUSTOMTESTS(mibnodes* ctnode);
void STOP_CUSTOMTESTS(mibnodes* ctnode);
void StringOidTest(mibtests* ctdef, char* oidtest);
void *CustomTestThread(void*);
void *DiagnoseThread(void*);
FILE* findpath (char* fileconf);

int RUN_DIAGNOSE_THREAD;
```

```
void READ_CONFIGS()
{
  FILE *config_nodes, *config_tests;

  char buffer[1024], command[255], error[1024];

  int  node_id, node_port;
  char node_ip[255], node_community[255];
  char test_name[255], test_oid[255], test_lcond[255], test_exec[255];
  int test_freq, state, test_node,  thisnode, count;

  mibnodes* nodeaux;
  mibtests* testaux;

  NT_table = GT_init_table(sizeof(mibnodes), NTfree, NULL);

  fprintf(stderr, "reading Timestamps.nodes.conf\n");
  config_nodes=findpath("Timestamps.nodes.conf");
  if (  (config_nodes = findpath("Timestamps.nodes.conf")) == NULL )
  {
    fprintf(stderr, "TIMESTAMPS ERROR: reading Timestamps.nodes.conf\n");
    return;
  }
  count = 1;

  while ( !feof(config_nodes) )
  {
    if ( fgets( buffer, sizeof(buffer), config_nodes) == 0 || buffer[0] == '#')
      continue;
    if ( sscanf(buffer, "%d %s %d %s\n",
                &node_id, node_ip, &node_port, node_community) < 4 )
      continue;
    nodeaux = GT_create(NT_table, node_id);
    nodeaux->index = node_id;
    nodeaux->port  = node_port;
    strcpy(nodeaux->community, node_community);
    nodeaux->TD_table = GT_init_table(sizeof(mibtests), TDfree, NULL);
    nodeaux->TD_table_updated = NODATA;
    pthread_mutex_init(&(nodeaux->mutexnode), NULL);
    nodeaux->ipaddress = iptoint(node_ip, error);
    gettimeofday ( &nodeaux->lastevent_time, NULL);
    memset( &nodeaux->ff_time, 0, sizeof(nodeaux->ff_time));
    memset( &nodeaux->fa_time, 0, sizeof(nodeaux->ff_time));
    if ( nodeaux->ipaddress == -1)
      continue;
  }
  fclose(config_nodes);
  count = 1;
  memset(buffer, 0, sizeof(buffer));
  state = NONE;
  thisnode   = getthisnodeid();
  TesterNode = GT_find(NT_table, thisnode);

  if (  (config_tests = findpath("Timestamps.tests.conf")) == NULL )
  {
    fprintf(stderr, "TIMESTAMPS ERROR: reading Timestamps.tests.conf\n");
    return;
  }
  else
  {
    TesterNode->TD_table_updated = UPDATED;
  }
```

```
while ( !feof(config_tests) )
{
  if ( fgets( buffer, sizeof(buffer), config_tests) == 0 || buffer[0] == '#')
    continue;
  sscanf(buffer, "%s", command);
  if ( strcmp(command, "TEST") == 0 )
  {
    sscanf(buffer, "%s %d", command, &test_node);
    if (test_node != thisnode )
    {
      state = NONE;
      continue;
    }
    state = TOKEN_TEST;
    testaux = GT_create(TesterNode->TD_table, count);
    testaux->index = count;
    count ++ ;
    testaux->type = 0;
    testaux->state = NONE;
  }

  if ( state != TOKEN_TEST )
    continue;

  if ( strcmp(command, "SERVICE") == 0 )
  {
    strncpy (testaux->name, buffer, strlen(buffer)-1);
    if ( (testaux->type & ( SERVICE | DEVICE )) != 0 )
      fprintf(stderr, "WARNING: redefinition of test %s\n", testaux->name);
    else
      testaux->type = testaux->type | SERVICE;
  }
  if ( strcmp(command, "DEVICE")  == 0 )
  {
    strncpy (testaux->name, buffer, strlen(buffer)-1);
    if ( (testaux->type & ( SERVICE | DEVICE )) != 0 )
      fprintf(stderr, "WARNING: redefinition of test %s\n", testaux->name);
    else
      testaux->type = testaux->type | DEVICE;
  }
  if ( strcmp(command, "SNMPGET")== 0 )
  {
    if ( (testaux->type & ( SNMPGET | EXECUTE )) != 0 )
      fprintf(stderr, "WARNING: type redefinition of test %s\n",
              testaux->name);
    else
      testaux->type = testaux->type | SNMPGET;
    if ( ParseOidTest( testaux, buffer, error) == FAULTYFREE )
      strcpy(testaux->def, buffer);
    else
      strcpy(testaux->def, error);
  }
  if ( strcmp(command, "EXECUTE") == 0 )
  {
    if ( (testaux->type & ( SNMPGET | EXECUTE )) != 0 )
      fprintf(stderr, "WARNING: type redefinition of test %s\n", testaux->name);
    else
      testaux->type = testaux->type | EXECUTE;
    sscanf(buffer, "%s %s", command, test_exec);
    strcpy(testaux->exec, test_exec);
    strncpy(testaux->def, buffer, strlen(buffer)-1);
  }
```

```
    if ( strcmp(command, "FREQ")    == 0 )
    {
      sscanf(buffer,"%s %d", command, &test_freq);
      testaux->freq = test_freq;
    }
  }
  fclose(config_tests);
}

int TestaNode( nodo* tester, int node,
               char *TESTED_State, char* TESTED_DIAGNOSE_LIST,
               unsigned* TESTED_TIMESTAMP_LIST)
{
  if ( node+1 > tester->n)
  {
    *TESTED_State = FAULTY;
  }
  else
  {
    tester->numtests++;
    GET_INFO (tester, node+1,
              TESTED_State, TESTED_DIAGNOSE_LIST, TESTED_TIMESTAMP_LIST);
  }
  return (int)(*TESTED_State);
}

void GET_INFO( nodo* tester, int node,
               char* TESTED_State, char* TESTED_DIAGNOSE_LIST,
               unsigned* TESTED_TIMESTAMP_LIST)
{

  mibnodes   * nodeaux;
  myresponse * mraux;
  ctgetinfo  * ctgiaux;
  mibtests   * ctdefaux;

  GT_table *MR_list;
  oid tsoid[] = {TIMESTAMPSOID};
  char ipaddress[255];
  char buffer[255], command[25];
  int  address, count, x, y, index, extra_test_node, extra_test_index;

  index = sizeof(tsoid)/sizeof(oid);
  nodeaux = GT_find(NT_table, node);
  address = nodeaux->ipaddress;
  sprintf(ipaddress, "%d.%d.%d.%d", (address >> 24) & 0xFF, (address >> 16) & 0xFF,
          (address >> 8) & 0xFF, (address & 0xFF) );

  if ( mybulkwalk(&MR_list, tsoid, sizeof(tsoid), ipaddress,
                  nodeaux->community, nodeaux->port) == 0 )
  {
    GT_Free (&MR_list);
    *TESTED_State      = FAULTY;
    if ( ping(ipaddress) == 0 )
      nodeaux->stateping = FAULTYFREE;
    else
      nodeaux->stateping = FAULTY;
    TESTED_PING_LIST[node-1]     = nodeaux->stateping;
    TESTED_DIAGNOSE_LIST[node-1] = FAULTY;
    return;
  }
```

```
*TESTED_State = FAULTY;
TESTED_DIAGNOSE_LIST[node-1] = FAULTY;
TESTED_PING_LIST[node-1] = FAULTYFREE;
for ( x=0; x< MR_list->data_count; x++)
{
  mraux = GT_get(MR_list, x);
  if ( mraux->name[index] == CTRLTABLE )
  {
    switch ( mraux->name[index+1] )
    {
    case CTRLCONTROL:
      if ( *mraux->val.integer != active )
        return;
      break;
    case CTRLSTATE:
      TESTED_DIAGNOSE_LIST[node-1] = *mraux->val.integer;
      *TESTED_State                = *mraux->val.integer;
      break;
    }
  }

  if ( mraux->name[index] == NTTABLE )
  {
    switch ( mraux->name[index+1])
    {
    case NTSTATE:
      switch ( mraux->name[index+2])
      {
      case NTSNMP:
        TESTED_DIAGNOSE_LIST[mraux->name[index+3]-1]=*mraux->val.integer;
        break;
      case NTPING:
        TESTED_PING_LIST[mraux->name[index+3]-1]=*mraux->val.integer;
        break;
      }
      break;

    case NTCUSTOMTEST:
      if ( mraux->name[index+2] != NTCTNAME)
      {
        ctgiaux =  GT_find( TESTED_CT_TABLE, mraux->name[index+3]);
        if ( ctgiaux != NULL )
          ctdefaux = GT_find (ctgiaux->TD_table, mraux->name[index+4]);
      }

      switch ( mraux->name[index+2])
      {
      case NTCTNAME:

        ctgiaux = GT_find( TESTED_CT_TABLE, mraux->name[index+3]);
        if ( ctgiaux == NULL )
        {
          ctgiaux = GT_create (TESTED_CT_TABLE, mraux->name[index+3]);
          ctgiaux->index = mraux->name[index+3];
          ctgiaux->TD_table = GT_init_table(sizeof(mibtests), TDfree, NULL);
        }
        ctdefaux = GT_find (ctgiaux->TD_table, mraux->name[index+4]);
        if ( ctdefaux == NULL )
        {
          ctdefaux = GT_create ( ctgiaux->TD_table, mraux->name[index+4]);
          ctdefaux->index = mraux->name[index+4];
        }
```

```
            strncpy(ctdefaux->name, mraux->val.string, mraux->val_len);

            if ( ctdefaux->name[0] == '(' )
            {
              sscanf(ctdefaux->name, "(%d.%d) %s", &extra_test_node, &extra_test_index, buffer);
            }
            else
            {
              sscanf (ctdefaux->name, "%s", buffer);
            }
            ctdefaux->type = 0;
            if ( strcmp(buffer, "DEVICE") == 0 )
              ctdefaux->type = ctdefaux->type | DEVICE;
            else if ( strcmp(buffer, "SERVICE") == 0 )
              ctdefaux->type = ctdefaux->type | SERVICE;

            break;

          case NTCTDEF:
            strncpy(ctdefaux->def, mraux->val.string, mraux->val_len);
            ctdefaux->def[mraux->val_len]=0;
            sscanf(ctdefaux->def, "%s", buffer);
            if ( strcmp(buffer, "SNMPGET") == 0 )
            {
              ctdefaux->type = ctdefaux->type | SNMPGET;
              strncpy(buffer, mraux->val.string, mraux->val_len);
              buffer[mraux->val_len]=0;
              ParseOidTest(ctdefaux, buffer, NULL);
            }
            if ( strcmp(buffer, "EXECUTE") == 0 )
            {
              ctdefaux->type = ctdefaux->type | EXECUTE;
              strncpy(buffer, mraux->val.string, mraux->val_len);
              buffer[mraux->val_len]=0;
              sscanf(buffer, "%s %s", command, ctdefaux->exec);
            }

            break;
          case NTCTFREQ:
            ctdefaux->freq = *mraux->val.integer;
            break;
          case NTCTSTATE:
            ctdefaux->state = *mraux->val.integer;
            break;
          }
          break;
        case NTTIMESTAMP:
          TESTED_TIMESTAMP_LIST[mraux->name[index+2]-1]= (unsigned)*((int*)mraux->val.integer);
          break;
        }
      }
    }
    GT_Free (&MR_list);
}


void NTfree(void* data)
{
  pthread_mutex_destroy(&((mibnodes*)data)->mutexnode);
  free ((mibnodes*)data);
```

```
}

void TDfree(void* data)
{
  free ((mibtests*)data);
}

void CTgetinfofree(void *data)
{
  free ((ctgetinfo*)data);
}


int ping(char *host)
{
  int status;
  if ( fork() == 0)
  {
    execlp("ping", "ping", host, "-c 3", "-q", 0);
  }
  else
  {
    wait(&status);
    if ( WIFEXITED(status) == 0 )
      return -1;
    return WEXITSTATUS(status);
  }
}

int execute(char *program)
{
  int status, pid;
  pid = fork();
  if( pid != 0 )  /* Parent */
  {
    pid = wait( &status );
    if ( WIFEXITED(status) == 0 )
      return -1;
    return WEXITSTATUS(status);
  }
  if ( pid == 0 )
  {
    execlp(program, program,0);
    exit(-1);
  }
}


int localips(struct in_addr ips[], int* count)
{
  char* ifname;
  int i;
  struct if_nameindex * ifs;
  int fd;
  struct ifreq if_info;

  ifs = if_nameindex();
  for (i = 0 ; ifs[i].if_name; i++)
  {
    ifname = ifs[i].if_name;
    memset(&if_info, 0, sizeof(if_info));
    strncpy(if_info.ifr_name, ifname, IFNAMSIZ-1);
```

```
    if ((fd=socket(AF_INET, SOCK_DGRAM, 0)) == -1)
      return -1;

    if (ioctl(fd, SIOCGIFADDR, &if_info) == -1)
    {
      close(fd);
      return -1;
    }
    memcpy(&ips[i], &((struct sockaddr_in *)&if_info.ifr_addr)->sin_addr,
          sizeof(struct in_addr));

  }
  if_freenameindex (ifs);
  close(fd);
  *count = i;
  return 0;
}

int getthisnodeid ()
{
  mibnodes* nodeaux;
  struct hostent *aux_ip;
  int ip, x, z, count;
  char localhost[255];
  struct in_addr ips[10];
  if ( localips(ips, &count) == -1)
  {
    fprintf(stderr, "TIMESTAMPS ERROR: reading localips\n");
    return 0;
  }
  for ( x=0; x<NT_table->data_count; x++)
  {
    nodeaux = GT_get( NT_table, x);
    for (z=0; z<count; z++)
    {
      if ( nodeaux->ipaddress == ntohl(ips[z].s_addr) )
        return nodeaux->index;
    }
  }
  return 0;
}

void START_CUSTOMTESTS(mibnodes* ctnode)
{
  int x;
  mibtests* ctdef;
  for( x=0; x<ctnode->TD_table->data_count; x++)
  {
    ctdef = GT_get(ctnode->TD_table, x);
    ctdef->thread = 0;
    ctdef->state = UNKNOWNSTATE;
    if ( ctdef->freq > 0 &&
         (ctnode->index == TesterNode->index || (ctdef->type & (SERVICE|DEVICE) ) == DEVICE ) )
    {
      pthread_create( &ctdef->thread, NULL, CustomTestThread, (void*)ctdef);
    }
  }
}


void STOP_CUSTOMTESTS(mibnodes* ctnode)
```

```
{
  int x;
  mibtests* ctdef;
  for( x=0; x<ctnode->TD_table->data_count; x++)
  {
    ctdef = GT_get(ctnode->TD_table, x);
    if( ctdef->thread != 0 )
      pthread_cancel(ctdef->thread);
    ctdef->thread = 0;
  }

}

int COUNT_TESTS(mibnodes* ctnode)
{
  int x;
  int count;
  mibtests* ctdef;
  count = 0;
  for( x=0; x<ctnode->TD_table->data_count; x++)
  {
    ctdef = GT_get(ctnode->TD_table, x);
    if ( ctnode == TesterNode || (ctdef->type & (SERVICE|DEVICE) ) == DEVICE )
      count++;
  }
  return count;
}

void *CustomTestThread(void* _ctdef)
{
  mibtests* ctdef;
  mibnodes* auxmibnode;
  myresponse* mraux;

  char ipaddress[255];

  int result;
  GT_table* MR_list;

  ctdef = _ctdef;

  /* conferir o teste de IP / conferir o teste de OID */
  while (1)
  {
    sleep(ctdef->freq);
    switch (ctdef->type & ( EXECUTE | SNMPGET ) )
    {
    case EXECUTE:
      if ( execute(ctdef->exec) == 0 )
      {
        ctdef->state = FAULTYFREE;
      }
      else
      {
        ctdef->state = FAULTY;
      }
      break;

    case SNMPGET:
      sprintf(ipaddress, "%d.%d.%d.%d",
              (ctdef->ip>> 24) & 0xFF, (ctdef->ip>> 16) & 0xFF,
              (ctdef->ip >> 8) & 0xFF, (ctdef->ip  & 0xFF) );
```

```
if ( myget(&MR_list, ctdef->ObjectID, ctdef->ObjectID_len,
           ipaddress, ctdef->community, ctdef->port) == 0)
{
  GT_Free(&MR_list);
  ctdef->state = FAULTY;
  continue;
}
else
{
  mraux = GT_get(MR_list, 0);
  switch(mraux->type)
  {
  case ASN_OCTET_STR:
    if ( ctdef->type_val != mraux->type )
    {
      ctdef->state = FAULTY;
      GT_Free(&MR_list);
      continue;
    }
    if ( mraux->val_len > ctdef->val_len )
      result = 1;
    else
      if ( mraux->val_len < ctdef->val_len )
        result = -1;
      else
        result = strncmp(mraux->val.string,
                         ctdef->val.string,
                         mraux->val_len);
           break;

  case ASN_COUNTER:
  case ASN_GAUGE:
  case ASN_TIMETICKS:
  case ASN_INTEGER:
    if ( ctdef->type_val != ASN_INTEGER )
    {
      ctdef->state = FAULTY;
      GT_Free(&MR_list);
      continue;
    }
    if ( *mraux->val.integer == *mraux->val.integer )
      result = 0;
    if ( *mraux->val.integer > *ctdef->val.integer )
      result = 1;
    if ( *mraux->val.integer < *ctdef->val.integer )
      result = -1;
    break;

  case ASN_UINTEGER:
    if ( ctdef->type_val != mraux->type )
    {
      ctdef->state = FAULTY;
      GT_Free(&MR_list);
      continue;
    }
    if ( *mraux->val.integer == *(unsigned*) mraux->val.integer )
      result = 0;
    if ( *mraux->val.integer > *(unsigned*) mraux->val.integer )
      result = 1;
    if ( *mraux->val.integer < *(unsigned*) mraux->val.integer )
      result = -1;
```

```
          break;

        case ASN_IPADDRESS:
          if ( ctdef->type_val != mraux->type )
          {
            ctdef->state = FAULTY;
            GT_Free(&MR_list);
            continue;
          }
          if ( *mraux->val.integer == *(unsigned*) mraux->val.integer )
            result = 0;
          if ( *mraux->val.integer > *(unsigned*) mraux->val.integer )
            result = 1;
          if ( *mraux->val.integer < *(unsigned*) mraux->val.integer )
            result = -1;
          break;

        case ASN_OBJECT_ID:
          if ( ctdef->type_val != mraux->type )
          {
            ctdef->state = FAULTY;
            GT_Free(&MR_list);
            continue;
          }
          result = snmp_oid_compare( (oid*) mraux->val.string,
                                     mraux->val_len/sizeof(oid),
                                     (oid*) ctdef->val.string,
                                     ctdef->val_len/sizeof(oid));
          break;
        }
        if ( result <  0)
          result = (ctdef->operator & LESS == LESS) ;
        if ( result == 0)
          result = (ctdef->operator & EQUAL) == EQUAL;
        if ( result >  0)
          result = (ctdef->operator & GREATER) == GREATER;
        ctdef->state = !result;
        GT_Free(&MR_list);
      }
      break;
    }
  }
}


void StringOidTest(mibtests* ctdef, char* oidtest)
{
  char string_oid[SPRINT_MAX_LEN], string_op[3], string_val[255], string_ip[255];

  switch (ctdef->type & ( EXECUTE | SNMPGET ) )
  {
    case EXECUTE:
      strcpy ( oidtest, "EXECUTE ");
      strcat ( oidtest, ctdef->exec);
      break;
    case SNMPGET:

      sprintf(string_ip, "%d.%d.%d.%d",
                (ctdef->ip >> 24) & 0xFF,
                (ctdef->ip >> 16) & 0xFF,
                (ctdef->ip >> 8) & 0xFF,
                (ctdef->ip & 0xFF) );
```

```
      sprint_objid(string_oid, ctdef->ObjectID, ctdef->ObjectID_len/sizeof(oid));

      if ( ctdef->operator == LESS )
        strcpy (string_op, "<" );
      if ( ctdef->operator == GREATER )
        strcpy (string_op, ">" );
      if ( ctdef->operator == EQUAL )
        strcpy (string_op, "==");
      if ( ctdef->operator == (LESS | GREATER))
        strcpy (string_op, "!=");
      if ( ctdef->operator == (GREATER | EQUAL))
        strcpy (string_op, ">=");
      if ( ctdef->operator == (LESS | EQUAL))
        strcpy (string_op, "<=");

      switch ( ctdef->type_val )
      {
      case ASN_INTEGER:
      case ASN_TIMETICKS:
        sprintf(string_val, "%d", *ctdef->val.integer);
        break;

      case ASN_UINTEGER:
        sprintf(string_val, "%u", *ctdef->val.integer);
        break;

      case ASN_OCTET_STR:
        strncpy(string_val, ctdef->val.string, ctdef->val_len);
        break;

      case ASN_IPADDRESS:
        sprintf(string_val, "%d.%d.%d.%d",
                (*ctdef->val.integer >> 24) & 0xFF,
                (*ctdef->val.integer >> 16) & 0xFF,
                (*ctdef->val.integer >> 8) & 0xFF,
                (*ctdef->val.integer & 0xFF) );
        break;
      case ASN_OBJECT_ID:
        sprint_objid(string_val, ctdef->val.objid, ctdef->val_len/sizeof(oid));
        break;
      }
      sprintf(oidtest, "SNMPGET %s %d %s %s %s %c %s",
              string_ip, ctdef->port,
              ctdef->community, string_oid,
              string_op, ctdef->type_set, string_val);
      break;
  }
}

/* IP PROT COMMUNITY  objectID type logical_condition value  */
int ParseOidTest (mibtests* testdef, char* oidtest, char* error)
{
  char **getParam;
  oid oid_aux[MAX_OID_LEN];
  int size_oid_aux;
  int int_aux;
  char _command[255];
  char _ip[255], _community[255], _oid[255], _operator[255], _value[255], _type;
  int _port;

  if ( sscanf(oidtest, "%s %s %d %s %s %s %c %s", _command,  _ip,
```

```
            &_port, _community, _oid, _operator, &_type, _value) < 7)
  {
    sprintf(error, "ERROR, usage: ipaddress port community oid logicaloperator valuetype value\n");
    return 1;
  }

  testdef->ip = iptoint(_ip, error);
  if ( testdef->ip == -1 )
    return 1;

  testdef->port = _port;
  strcpy (testdef->community, _community);

  size_oid_aux = MAX_OID_LEN;
  if (!read_objid( _oid, oid_aux, &size_oid_aux) )
  {
    snmp_perror(_oid);
    return 1;
  }
  testdef->ObjectID = (oid*) malloc (size_oid_aux * sizeof(oid));
  testdef->ObjectID_len = size_oid_aux * sizeof(oid);
  memcpy ( testdef->ObjectID, oid_aux, size_oid_aux*sizeof(oid));

  testdef->operator = -1;
  if ( strcmp (_operator, "<" ) == 0 )
    testdef->operator = LESS ;
  if ( strcmp (_operator, ">" ) == 0 )
    testdef->operator = GREATER;
  if ( strcmp (_operator, "==") == 0 )
    testdef->operator = EQUAL;
  if ( strcmp (_operator, "!=") == 0 )
    testdef->operator = LESS | GREATER;
  if ( strcmp (_operator, ">=") == 0 )
    testdef->operator = GREATER | EQUAL;
  if ( strcmp (_operator, "<=") == 0 )
    testdef->operator = LESS | EQUAL;

  if ( testdef->operator == -1 )
  {
    sprintf(error, "ERROR, operator are: < > == != >= <=! \n");
    return 1;
  }

/*
  i INTEGER
  u UNSIGNED
  s STRING
  o OBJID
  t  TIMETICKS
  a  IPADDRESS
*/

  if ( _type!='i' &&  _type!='u' &&
       _type!='s' &&  _type!='o' &&
       _type!='t' &&  _type!='a')
  {
    sprintf(error, "ERROR: the known types are: i, u, s, o, t, a\n");
    return 1;
  }
  testdef->type_set = _type;

  switch (_type)
```

```
  {
  case 'i':
    testdef->type_val = ASN_INTEGER;
    testdef->val.integer = (long*) malloc (sizeof(long));
    int_aux = atoi(_value);
    memcpy (testdef->val.integer, &int_aux, sizeof(long));
    break;

  case 'u':
    testdef->type_val = ASN_UINTEGER;
    testdef->val.integer = (long*) malloc (sizeof(long));
    int_aux = atoi(_value);
    memcpy (testdef->val.integer, &int_aux, sizeof(long));
    break;

  case 't':
    testdef->type_val = ASN_TIMETICKS;
    testdef->val.integer = (long*) malloc (sizeof(long));
    int_aux = atoi(_value);
    memcpy (testdef->val.integer, &int_aux, sizeof(long));
    break;

  case 's':
    testdef->type_val = ASN_OCTET_STR;
    testdef->val.string = (char*) malloc (strlen(_value)*sizeof(char));

    /* pelo certo devia ser -1 no len... mas depois eu testo isto */
    memcpy ( testdef->val.string, _value, strlen(_value)*sizeof(char));
    testdef->val_len = strlen(_value)*sizeof(char);
    break;

  case 'a':
    testdef->type_val = ASN_IPADDRESS;
    testdef->val.integer = (long*) malloc (sizeof(long));
    int_aux = iptoint(_value, error);
    memcpy (testdef->val.integer, &int_aux, sizeof(long));
    if ( *testdef->val.integer == -1 )
    {
      return 1;
    }
    break;

  case 'o':
    testdef->type_val = ASN_OBJECT_ID;
    if ( !snmp_parse_oid( _value, oid_aux, &size_oid_aux))
    {
      fprintf(stderr, "ERROR: OID \"%s\" INVALID\n", _oid);
      return 1;
    }
    testdef->val.objid = (oid*) malloc (size_oid_aux * sizeof(oid));
    memcpy ( testdef->val.objid, oid_aux, size_oid_aux * sizeof(oid));
    break;
  }
  return 0;
}

int iptoint(char* _ip, char* error)
{
  int ip;
  struct hostent *aux_ip;
  if ( (ip = inet_addr(_ip)) == -1 )
  {
```

```
    aux_ip = gethostbyname(_ip);
    if ( aux_ip == NULL)
    {
      if (error!=NULL)
        sprintf(error, "ERROR: Unknown address: \"%s\"\n", _ip);
      return -1;
    }
    else
    {
      memcpy((char*)&ip, (char*)(aux_ip->h_addr), 4);
      ip = ntohl(ip);
    }
  }
  return ip;
}

pthread_t thread_timestamps;


void *RestartDiagnoseThread()
{
  int *ret;
  fprintf(stderr, "stopping timestamps\n");
  RUN_DIAGNOSE_THREAD = FALSE;
  pthread_join(thread_timestamps, (void**) &ret);
  READ_CONFIGS();
  RUN_TIMESTAMPS();
  pthread_exit(0);
}

void STOP_TIMESTAMPS()
{
  RUN_DIAGNOSE_THREAD = FALSE;
}

void RESTART_TIMESTAMPS()
{
  pthread_t thread_aux;
  int x;
  pthread_create( &thread_aux, NULL, RestartDiagnoseThread, (void*)&x);
  pthread_detach( thread_aux );
}



void RUN_TIMESTAMPS()
{
  int thisid, x;
  mibnodes* aux;
  InitTestedNodes( NT_table->data_count, 2 );
  ThisNode = NodosTestes (TesterNode->index-1, NT_table->data_count, 2);

  for (x=0; x<NT_table->data_count; x++)
  {
    aux = GT_get ( NT_table, x);
    aux->statesnmp = ThisNode->DIAGNOSE_LIST[aux->index-1];
    aux->timestamp = ThisNode->TIMESTAMP_LIST[aux->index-1];
  }
  START_CUSTOMTESTS(TesterNode);
  RUN_DIAGNOSE_THREAD = TRUE;
  CT_ON_LIST = (char*) malloc (sizeof(char)*NT_table->data_count);
  memset(CT_ON_LIST, FALSE, sizeof(char)*NT_table->data_count);
```

```
  CT_ON_LIST[ThisNode->id] = TRUE;
  pthread_create( &thread_timestamps, NULL, DiagnoseThread, (void*) ThisNode);
}

void *DiagnoseThread(void* param)
{
  int x,y,count,N;
  mibnodes* auxmibnode;
  ctgetinfo  *ctgiaux;
  mibtests* ctdefaux;
  GT_table *freeaux;
  nodo* ThisNode;
  struct timeval now;
  int statenode;

  ThisNode = param;
  N = NT_table->data_count;

  gettimeofday (&init_time, NULL);

  while(RUN_DIAGNOSE_THREAD == TRUE)
  {
    sleep(timestamp_interval);
    TESTED_PING_LIST = (char*) malloc(sizeof(char)*NT_table->data_count);
    TESTED_CT_TABLE =  GT_init_table(sizeof(ctgetinfo), CTgetinfofree, NULL);

    Rodada(ThisNode);
    statenode = TesterNode->statesnmp;
    for (x=0; x<N; x++)
    {
      auxmibnode = GT_find ( NT_table, x+1);
      pthread_mutex_lock ( &(auxmibnode->mutexnode) );
      ctgiaux = GT_find (TESTED_CT_TABLE, auxmibnode->index);
      if ( ctgiaux != NULL )
        freeaux = ctgiaux->TD_table;
      else
        freeaux = NULL;

      if ( UPDATED_INFO[auxmibnode->index-1] == TRUE || auxmibnode->TD_table_updated != UPDATED )
      {
        if ( UPDATED_INFO [ auxmibnode->index-1] == TRUE)
        {
          /* mutex para evitar o acesso dos gets as informacoes deste node */
          if ( auxmibnode->timestamp < ThisNode->TIMESTAMP_LIST[auxmibnode->index-1]
               && auxmibnode->statesnmp != ThisNode->DIAGNOSE_LIST[auxmibnode->index-1])
          {
           // gettimeofday ( &now, NULL);
            if ( auxmibnode->statesnmp == FAULTY)
            {
              auxmibnode->fa_time.tv_sec += ( now.tv_sec - auxmibnode->lastevent_time.tv_sec);
             // gettimeofday ( &auxmibnode->lastevent_time, NULL);
            }
            if ( ThisNode->DIAGNOSE_LIST[auxmibnode->index-1] == FAULTY )
            {
              auxmibnode->ff_time.tv_sec += ( now.tv_sec - auxmibnode->lastevent_time.tv_sec);
              //gettimeofday ( &auxmibnode->lastevent_time, NULL);
            }
          }
          auxmibnode->statesnmp = ThisNode->DIAGNOSE_LIST [auxmibnode->index-1];
          auxmibnode->timestamp = ThisNode->TIMESTAMP_LIST[auxmibnode->index-1];
          auxmibnode->stateping = TESTED_PING_LIST[auxmibnode->index-1];
        }
```

```
      if ( auxmibnode->index-1 != ThisNode->id)
      {
        /* Custom Tests Update */
        if ( ctgiaux != NULL  && CT_ON_LIST[auxmibnode->index-1] == FALSE )
        {
          freeaux = auxmibnode->TD_table;
          auxmibnode->TD_table = ctgiaux->TD_table;
          auxmibnode->TD_table_updated = UPDATED;
        }
        if ( auxmibnode->statesnmp == FAULTY )
          removeextratests(auxmibnode->index);
      }
    }
    if ( freeaux != NULL )
      GT_Free (&freeaux);
    pthread_mutex_unlock ( &(auxmibnode->mutexnode) );
  }

  GT_Free (&TESTED_CT_TABLE);
  free (TESTED_PING_LIST);


  if ( ThisNode->state != FAULTYFREE)
    continue;

  /* Determines the "custom test" tasks  to be assumed or abandoned by this node */
  for ( x=ThisNode->id+1;
        x<ThisNode->id+N && ThisNode->DIAGNOSE_LIST[x%N]==FAULTY;
        x++ )
  {
    /* assume task */
    if ( CT_ON_LIST[x%N] == FALSE )
    {
      auxmibnode = GT_find ( NT_table, ((x%N)+1));
      if ( auxmibnode->TD_table_updated != NODATA)
      {
        auxmibnode->TD_table_updated = UPDATED;
        CT_ON_LIST[x%N] = TRUE;
        START_CUSTOMTESTS ( auxmibnode );
      }
    }
  }

  /* leave the task only if the nodo is FAULTYFREE
     wouldn't be FAULTYFREE_INITING*/
  if ( ThisNode->DIAGNOSE_LIST[x%N] == FAULTYFREE)
  {
    for (;CT_ON_LIST[x%N] == TRUE && x%N != ThisNode->id; x++)
    {
      CT_ON_LIST[x%N] = FALSE;
      auxmibnode = GT_find ( NT_table, ((x%N)+1));
      STOP_CUSTOMTESTS (auxmibnode);
    }
  }
}

fprintf(stderr, "limpando\n");
/* let's stop the custom tests */
for ( x=0; x<N; x++ )
{
  if ( CT_ON_LIST[x%N] = TRUE )
  {
```

```
        auxmibnode = GT_find ( NT_table, x+1);
        STOP_CUSTOMTESTS (auxmibnode);
      }
    }
    free ( CT_ON_LIST);
    FreeTestedNodes( NT_table->data_count, 2 );
    FreeNodosTestes( &ThisNode);

    for (x=0; x<NT_table->data_count; x++)
    {
      auxmibnode = GT_find (NT_table, x+1);
      count = auxmibnode->TD_table->data_count;
      /* verificar este count */
      for (y=1; y<=count; y++)
      {
        ctdefaux = GT_find(auxmibnode->TD_table, y);
        if ( ctdefaux!=NULL )
        {
          freectval (ctdefaux);
          GT_destroy (auxmibnode->TD_table, y);
        }
      }
      GT_Free(&auxmibnode->TD_table);
    }
    GT_Free(&NT_table);
    fprintf(stderr,"terminei");
    pthread_exit(0);
}

void freectval(mibtests* auxct)
{
    if (auxct != NULL )
    {
      free (auxct->ObjectID);
      switch ( auxct->type_val)
      {
      case ASN_COUNTER:
      case ASN_GAUGE:
      case ASN_TIMETICKS:
      case ASN_INTEGER:
      case ASN_UINTEGER:
      case ASN_IPADDRESS:
        free ( auxct->val.integer);
        break;
      case ASN_OCTET_STR:
        free ( auxct->val.string);
        break;
      case ASN_OBJECT_ID:
        free ( auxct->val.objid);
        break;
      }
    }
}

void removeextratests(int node)
{
    int x,  count;
    mibnodes* auxmibnode;
    mibtests* auxct;
    auxmibnode = GT_find(NT_table, node);
    count = auxmibnode->TD_table->data_count;
    /* verificar este count */
```

```
  for (x=1; x<=count; x++)
  {
    auxct = GT_find(auxmibnode->TD_table, x);
    if ( auxct!=NULL && auxct->name[0]=='(' )
    {
      freectval (auxct);
      GT_destroy (auxmibnode->TD_table, x);
    }
  }
}

FILE* findpath (char* fileconf)
{
  char configfile[300];
  char *envconfpath, *homepath;
  char *cptr1, *cptr2;
  char defaultPath[SPRINT_MAX_LEN];
  int config_errors;
  FILE* conf;
  config_errors = 0;
  if ((envconfpath = getenv("SNMPCONFPATH")) == NULL)
  {
    homepath=getenv("HOME");
    sprintf(defaultPath,"%s%c%s%c%s%c%s%s%c%s%c%s",
            SNMPCONFPATH, ENV_SEPARATOR_CHAR,
            SNMPSHAREPATH, ENV_SEPARATOR_CHAR, SNMPLIBPATH,
            ENV_SEPARATOR_CHAR,
            ((homepath == NULL) ? "" : homepath),
            ((homepath == NULL) ? "" : "/.snmp"),
            ENV_SEPARATOR_CHAR,
            "/etc/snmp",
            ENV_SEPARATOR_CHAR, PERSISTENT_DIRECTORY);
    free(homepath);
    envconfpath = defaultPath;
  }
  envconfpath = (char*)strdup(envconfpath);
  cptr1 = cptr2 = envconfpath;
  conf = NULL;
  while(*cptr1 != 0 && conf == NULL )
  {
    do{
      cptr1++;
    }while(*cptr1 != 0 && *cptr1 != ENV_SEPARATOR_CHAR);
    strncpy(configfile, cptr2, cptr1-cptr2);
    configfile[cptr1-cptr2]=0;
    cptr2=cptr1+1;
    strcat(configfile, "/");
    strcat(configfile, fileconf);
    conf = fopen(configfile, "r" );
  }

  free(envconfpath);
  return conf;
}
```

```
#include <math.h>
#include <stdio.h>
#include <memory.h>

#include "listas.h"
#include "clusteri.h"


// #define DEBUG


nodos** AllTestedNodes;
unsigned char** ALL_DISTANCE;


nodos* _TestedNodes(int tester, int n, int i);
void   _NodosDistance(int n, int i);
nodos* _MoreInfo(nodo* tester, int p);
void   _UpdateTimestamps(nodo* tester,
                         char* TESTED_State,
                         char* TESTED_DIAGNOSE_LIST,
                         unsigned * TESTED_TIMESTAMP_LIST);

nodos* _TestedNodes(int tester, int n, int i)
{
  int x, _round, round, aux;
  int tInt, tExt;
  nodos* testslist;

  /* obtain tester tests */
  _round = ceil(log(n)/log(i));
  testslist = CriaLista();
  if ( tester > n ) return testslist;
  tInt = 1; tExt = i;
  for ( round=1; round<=_round; round++)
  {
    for ( x=0; x<i; x++)
    {
      aux = floor(tester/tExt)*tExt + tester%tInt + tInt*x;
      if ( aux!=tester) AddList(testslist, aux);
    }
    tInt = tInt * i; tExt = tExt * i;
  }
  return testslist;
}

void InitTestedNodes(int n, int i)
{
  int x, N;
  N = pow (i, ceil(log(n)/log(i)));
  AllTestedNodes = (nodos**) malloc ( sizeof(nodos*)*N);
  for ( x=0; x<N; x++)
    AllTestedNodes[x]=_TestedNodes(x, n, i);
  _NodosDistance(n, i);
  UPDATED_INFO = (unsigned char*) malloc ( sizeof(char) * N);
  memset (UPDATED_INFO, FALSE, sizeof(char)*N);
}

void FreeTestedNodes(int n, int i)
```

```
{
  int x;
  int N;
  N = pow (i, ceil(log(n)/log(i)));

  for ( x=0; x<N; x++)
    FreeList(&AllTestedNodes[x]);
  free(AllTestedNodes);

  for (x=0; x<n; x++)
    free ( ALL_DISTANCE[x] );
  free(ALL_DISTANCE);

  free(UPDATED_INFO);
}

nodos* TestedNodes(int tester)
{
  return AllTestedNodes[tester];
}

void FreeNodosTestes(nodo** nodotester )
{
  int x;
  free ( (*nodotester)->UNKNOWN_LIST );
  free ( (*nodotester)->BLOCKED_LIST );
  free ( (*nodotester)->DIAGNOSE_LIST );
  free ( (*nodotester)->TIMESTAMP_LIST );
  for ( x=0; x<(*nodotester)->nrounds; x++)
  {
    FreeList ( &(*nodotester)->tests[x] );
  }
  free (*nodotester);
}

nodo* NodosTestes(int tester, int n, int i)
{
  int N, x, _round, round, aux;
  int tInt, tExt;
  nodos* testslist;
  nodo* newnode;


#ifdef DEBUG
  fprintf(stderr, "\nINITING NODE %d", tester);
#endif

 /* obtain tester tests */
  _round = ceil(log(n)/log(i));
  N = pow (i, _round);

  /* fill newnode structure */
  newnode = (nodo*) malloc (sizeof(nodo));
  newnode->id = tester; newnode->round = 0; newnode->power = i;
  newnode->roundcount=0;
  newnode->numtests=0;
  newnode->n = n;
  newnode->N = N;
  newnode->state = FAULTYFREE_INITING;
  newnode->UNKNOWN_LIST   = (char*) malloc (sizeof(char)*N);
  newnode->BLOCKED_LIST   = (char*) malloc (sizeof(char)*N);
  newnode->DIAGNOSE_LIST  = (char*) malloc (sizeof(char)*N);
```

```
  newnode->TIMESTAMP_LIST = (unsigned*) malloc (sizeof(unsigned)*N);

  memset(newnode->UNKNOWN_LIST,     UNKNOWN, N);
  memset(newnode->DIAGNOSE_LIST+n, KNOWN, N-n);
  memset(newnode->DIAGNOSE_LIST,   FAULTYFREE_INITING, N);
  memset(newnode->DIAGNOSE_LIST+n, FAULTY, N-n);
  memset(newnode->BLOCKED_LIST,     UNBLOCKED, N);
  memset(newnode->TIMESTAMP_LIST,  0, N*sizeof(unsigned));

  newnode->nrounds = _round;

  tInt = 1; tExt = i;
  for ( round=1; round<=_round; round++)
  {
    testslist = CriaLista();
    for ( x=0; x<i; x++)
    {
      aux = floor(tester/tExt)*tExt + tester%tInt + tInt*x;
      if ( aux!=tester)
        AddList(testslist, aux);
    }
    newnode->tests[round-1]=testslist;
    tInt = tInt*i;
    tExt = tExt*i;
  }

  return newnode;
}


void _NodosDistance(int n, int i)
{
  int N, x, _round, round, aux, nodeid;
  int tInt, tExt;
  nodos* testslist;

  ALL_DISTANCE = (unsigned char**) malloc (sizeof(char*)*n);
  _round = ceil(log(n)/log(i));
  N = pow (i, _round);

  for ( nodeid=0; nodeid < n; nodeid++)
  {
    ALL_DISTANCE[nodeid] = (unsigned char*) malloc (sizeof(char)*N);
    memset(ALL_DISTANCE[nodeid], 0, sizeof(char)*N);

    for ( tInt=1, tExt=i, round=1; round<=_round; round++)
    {
      for ( x=0; x<i; x++)
      {
        aux = floor(nodeid/tExt)*tExt + nodeid%tInt + tInt*x;
        if ( aux!=nodeid)
          ALL_DISTANCE[nodeid][aux]=1;
      }
      tInt = tInt*i;
      tExt = tExt*i;
    }

    /* distance list fill up */
    for ( round=1; round<=_round; round++)
      for ( x=0 ; x<n; x++)
        if ( ALL_DISTANCE[nodeid][x] == round )
        {
```

```
          testslist = TestedNodes(x);
          for ( testslist->aux=testslist->cabeca; testslist->aux!=NULL; testslist->aux=testslist->aux->next)
          {

            if ( ( ALL_DISTANCE[nodeid][testslist->aux->num] == 0
                   || ALL_DISTANCE[nodeid][testslist->aux->num] == round+1 )
                 && testslist->aux->num!=nodeid)
              ALL_DISTANCE[nodeid][testslist->aux->num] = round+1;
          }
        }

#ifdef DEBUG
    for ( round=1; round<=_round; round++)
    {
      fprintf(stderr, "\nDISTANCIA %d\n", round);
      for (x=0;x<N; x++)
        if (ALL_DISTANCE[nodeid][x]==round) fprintf(stderr, " %2d ", x);
    }
    fprintf(stderr, "\n");
#endif

  }
}

nodos* _MoreInfo(nodo* tester, int p)
{
  int x;
  nodos *RESULT_LIST;
  RESULT_LIST = CriaLista();
  if ( p >= tester->n) return RESULT_LIST;
  for ( x = 0; x<tester->N; x++)
    if ( ALL_DISTANCE[p][x] < ALL_DISTANCE[tester->id][x] )
      AddList(RESULT_LIST, x);
  DelList(RESULT_LIST, p);
  return RESULT_LIST;
}


nodos* unreacheable(nodo* tester, nodos* needinfo)
{
  nodos *RESULT_LIST,  *TESTED, *AUX;
  int cont_faulty;

  RESULT_LIST = CriaLista();
  TESTED = TestedNodes(tester->id);
  cont_faulty = 0;
  for ( TESTED->aux=TESTED->cabeca; TESTED->aux != NULL; TESTED->aux = TESTED->aux->next )
    if ( tester->DIAGNOSE_LIST[TESTED->aux->num] == FAULTY)
      cont_faulty++;
  if ( cont_faulty <= 1) return RESULT_LIST;

  for ( needinfo->aux=needinfo->cabeca; needinfo->aux != NULL; needinfo->aux = needinfo->aux->next)
  {
    /* if it is in blocked, its already been in the test list */
    if ( tester->BLOCKED_LIST[needinfo->aux->num] != BLOCKED )
      AddList(RESULT_LIST, needinfo->aux->num);
  }
  for ( TESTED->aux=TESTED->cabeca; TESTED->aux != NULL && RESULT_LIST->cabeca!=NULL; TESTED->aux = TESTED->aux->n
  {
    if ( tester->DIAGNOSE_LIST[TESTED->aux->num] == FAULTYFREE )
    {
      AUX = _MoreInfo(tester, TESTED->aux->num);
```

```
      for ( AUX->aux = AUX->cabeca; AUX->aux != NULL; AUX->aux = AUX->aux->next )
        DelList(RESULT_LIST, AUX->aux->num);
      FreeList(&AUX);
    }
  }
  return RESULT_LIST;
}


void _UpdateTimestamps(nodo*tester,
                       char* TESTED_State, char* TESTED_DIAGNOSE_LIST, unsigned* TESTED_TIMESTAMP_LIST)
{
  int x, any_body_stable;
  char *diagnose, *unknown, *blocked;
  unsigned  *timestamp;
  diagnose = tester->DIAGNOSE_LIST;
  unknown =  tester->UNKNOWN_LIST;
  timestamp =tester->TIMESTAMP_LIST;
  blocked =  tester->BLOCKED_LIST;

  /* if it is FAULTFREE, then take the biggest timestamps */
  if ( *TESTED_State == FAULTYFREE && tester->state == FAULTYFREE_INITING)
  {
#ifdef DEBUG
    fprintf(stderr, "    UPDATING TIMESTAMPS\n" );
#endif
    for (x=0; x<tester->n; x++)
    {
      if ( unknown[x] == UNKNOWN)
      {
        diagnose [x]=TESTED_DIAGNOSE_LIST [x];
        timestamp[x]=TESTED_TIMESTAMP_LIST[x];
        UPDATED_INFO[x] = TRUE;
        unknown[x]=KNOWN;
      }
    }
    UPDATED_INFO[tester->id] = TRUE;
    tester->state = FAULTYFREE;
    if ( diagnose[tester->id] == FAULTY )
      timestamp[tester->id]=timestamp[tester->id]+1;
    diagnose[tester->id] = FAULTYFREE;
  }

  if ( tester->state == FAULTYFREE_INITING && *TESTED_State != FAULTYFREE &&
       tester->roundcount > (tester->nrounds * tester->nrounds))
  {
#ifdef DEBUG
    fprintf(stderr, "    UPDATING TIMESTAMPS\n" );
#endif
    any_body_stable = FALSE;
    for (x=0; x<tester->n; x++)
      if ( diagnose[x] == FAULTYFREE )
        any_body_stable = TRUE;

    if ( any_body_stable == FALSE )
    {
      for ( x=0; x<tester->n; x++)
      {
        UPDATED_INFO[x] = TRUE;
        if ( tester->DIAGNOSE_LIST[x] == FAULTY )
          timestamp[x] = 1;
        else
```

```
          timestamp[x] = 0;
        }
        tester->state = FAULTYFREE;
        memset( unknown, KNOWN, tester->N);
        tester->DIAGNOSE_LIST[tester->id] = FAULTYFREE;
        return;
      }
    }
}

void ReadInfo (nodo *tester, nodos* getinfo, nodos* blockedlist,
               char *TESTED_State, char* TESTED_DIAGNOSE_LIST, unsigned* TESTED_TIMESTAMP_LIST)
{
  char *diagnose, *unknown, *blocked;
  unsigned  *timestamp;
  int x;
#ifdef DEBUG
  fprintf(stderr, "    READINFO");
#endif
  diagnose = tester->DIAGNOSE_LIST;  unknown = tester->UNKNOWN_LIST;
  timestamp = tester->TIMESTAMP_LIST; blocked = tester->BLOCKED_LIST;
  for ( getinfo->aux = getinfo->cabeca; getinfo->aux != NULL; getinfo->aux = getinfo->aux->next )
  {
    if ( TESTED_TIMESTAMP_LIST[getinfo->aux->num] >= timestamp[getinfo->aux->num] )
    {
 #ifdef DEBUG
      fprintf(stderr, "ESTOU LENDO INFO\n");
      fprintf(stderr, " %3d ", getinfo->aux->num);
 #endif
      diagnose [getinfo->aux->num]=TESTED_DIAGNOSE_LIST[getinfo->aux->num];
      timestamp[getinfo->aux->num]=TESTED_TIMESTAMP_LIST[getinfo->aux->num];
      blocked[getinfo->aux->num]=UNBLOCKED;
      DelList(blockedlist, getinfo->aux->num);
      unknown[getinfo->aux->num]=KNOWN;
      UPDATED_INFO[getinfo->aux->num]=TRUE;
    }
  }

  for (x=0; x<tester->n; x++)
  {
    if (timestamp[x] < TESTED_TIMESTAMP_LIST[x])
    {
      diagnose [x]=TESTED_DIAGNOSE_LIST [x];
      timestamp[x]=TESTED_TIMESTAMP_LIST[x];
      UPDATED_INFO[x] = TRUE;
    }
  }
  if ( diagnose[tester->id] == FAULTY )
    timestamp[tester->id]=timestamp[tester->id]+1;
  diagnose[tester->id] = tester->state;

#ifdef DEBUG
  fprintf(stderr, "\n" );
#endif
}

void Rodada(nodo* tester)
{
  int result, lastresult, x;
  int tInt, firstnode, lastnode;
  int testnode, in_more;
```

```
  nodos *list, *tests, *blocked;

  char* TESTED_DIAGNOSE_LIST;
  unsigned*  TESTED_TIMESTAMP_LIST;
  char  *TESTED_State;

  TESTED_DIAGNOSE_LIST =  (unsigned char*)malloc (sizeof(char)*tester->N);
  TESTED_TIMESTAMP_LIST = (unsigned*) malloc (sizeof(unsigned)*tester->N);
  TESTED_State = (char*) malloc (sizeof(char));

  memset ( TESTED_DIAGNOSE_LIST,  0, sizeof(char)*tester->N);
  memset ( TESTED_TIMESTAMP_LIST, 0, sizeof(unsigned)*tester->N);
  memset ( UPDATED_INFO, FALSE, sizeof(char)*tester->N);

  blocked = CriaLista();

  tester->round++;
  tester->roundcount++;
  if ( tester->round > tester->nrounds ) tester->round = 1;
  tests = tester->tests[tester->round-1];
  tInt = (int) pow (tester->power, tester->round-1);

#ifdef DEBUG
  fprintf(stderr, "NODE %d \n    BLOCKED:", tester->id);
  for (x=0; x<tester->n; x++)
    if (tester->BLOCKED_LIST[x]==BLOCKED)
      fprintf(stderr, " %3d ", x);
  fprintf(stderr, "\n");
#endif

  in_more = FALSE;

  for ( tests->aux=tests->cabeca; tests->aux!=NULL;  tests->aux=tests->aux->next)
  {
    testnode = tests->aux->num;
    UPDATED_INFO[testnode] = TRUE;
    if ( in_more == TRUE && !FindList(blocked, testnode) )
      continue;

#ifdef DEBUG
    fprintf(stderr, "    TESTED %3d: ", testnode);
#endif

    lastresult = tester->DIAGNOSE_LIST[testnode];
    result   = TestaNode ( tester, testnode, TESTED_State, TESTED_DIAGNOSE_LIST, TESTED_TIMESTAMP_LIST);
    tester->DIAGNOSE_LIST[testnode]=result;
    if ( lastresult != result && ( lastresult == FAULTY || result == FAULTY) )
        tester->TIMESTAMP_LIST[testnode]++;

    _UpdateTimestamps(tester, TESTED_State, TESTED_DIAGNOSE_LIST, TESTED_TIMESTAMP_LIST);
    tester->BLOCKED_LIST[testnode] =UNBLOCKED;
    DelList(blocked, testnode);


#ifdef DEBUG
    if ( result == FAULTY )
      fprintf(stderr,  "FAULTY\n");
    if ( result == FAULTYFREE )
      fprintf(stderr,  "FAULTYFREE\n");
    if ( result == FAULTYFREE_INITING )
      fprintf(stderr,  "FAULTYFREE INITING\n");
#endif
```

```
    switch (result)
    {
    case FAULTYFREE:
    case FAULTYFREE_INITING:
      list = _MoreInfo(tester, testnode);
#ifdef DEBUG
      PrintList("    MORE_INFO", list);
#endif
      ReadInfo( tester,list, blocked, TESTED_State,
                TESTED_DIAGNOSE_LIST, TESTED_TIMESTAMP_LIST);
      FreeList(&list);
      break;
    case FAULTY:
      if ( in_more == FALSE)
      {
        firstnode = floor(testnode/tInt)*tInt;
        lastnode  = firstnode+tInt - 1;
        for ( x=firstnode; x<=lastnode; x++)
          AddList(blocked, x);
        DelList(blocked, testnode);
      }
      break;
    }

    if ( tests->aux->next == NULL && in_more == FALSE && blocked->cabeca!=NULL )
    {
      tests = CriaLista();
      AddList(tests, 0);
      for ( blocked->aux=blocked->cabeca;  blocked->aux!=NULL; blocked->aux= blocked->aux->next )
        if ( tester->BLOCKED_LIST[blocked->aux->num] == BLOCKED )
            AddList(tests, blocked->aux->num);
      tests->aux = tests->cabeca;
      in_more = TRUE;
    }

  }

  for ( blocked->aux=blocked->cabeca;  blocked->aux!=NULL;
        blocked->aux= blocked->aux->next )
    tester->BLOCKED_LIST[blocked->aux->num]=BLOCKED;

  if ( in_more == TRUE )
    FreeList(&tests);

  free (TESTED_DIAGNOSE_LIST);
  free (TESTED_TIMESTAMP_LIST);
  free (TESTED_State);

#ifdef DEBUG
  fprintf(stderr, "    BLOCKED:");
  for (x=0; x<tester->n; x++)
    if (tester->BLOCKED_LIST[x]==BLOCKED)
      fprintf(stderr, " %3d ", x);
  fprintf(stderr, "\n\n");
#endif

  FreeList(&blocked);
}
```