VINICIUS KWIECIEN RUOSO

UMA ESTRATÉGIA DE TESTES LOGARÍTMICA PARA O ALGORITMO HI-ADSD

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. Dr. Elias P. Duarte Jr.
Co-Orientador: Prof. Dr. Luis C. E. Bona

CURITIBA

2013

R944e Ruoso, Vinícius Kwiecien
Uma estratégia de testes logarítmica para o algoritmo Hi-ADSD [manuscrito] / Vinicius Kwiecien Ruoso. – Curitiba, 2013.
49f. : il. [algumas color.] ; 30 cm.

Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Ciência Exatas, Programa de Pós-graduação em Informática, 2013.

Orientador: Elias Procópio Duarte Jr. -- Co-orientador: Luis Carlos Erpen de Bona .

1. Logaritmos. 2. Algoritmo. I. Universidade Federal do Paraná. II. Duarte Jr. Elias Procópio. III Bona, Luis Carlos Erpen de. IV. Título.

CDD: 512.922

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Vinícius Kwiecien Ruoso, avaliamos o trabalho intitulado, *"Uma Estratégia de Testes Logarítmica para o Algoritmo Hi-ADSD"*, cuja defesa foi realizada no dia 24 de maio de 2013, às 09:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 24 de maio de 2013.

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

Prof. Dr. Luis Carlos Erpen De Bona
DINF/UFPR - Coorientador

Prof. Dr. Bogdan Tomoyuki Nassu
UTFPR - Membro Externo

Prof. Dr. Roverli Pereira Ziwich
TRT - Membro Externo

Prof. Dr. Andrea Weber
PPGINF/UFPR - Membro Interno

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

SUMÁRIO

LISTA DE SIGLAS E ABREVIAÇÕES

Adaptive-DSD	*Adaptive Distributed System-Level Diagnosis Algorithm*
API	*Application Programming Interface*
DHT	*Distributed Hash Table*
DiVHA	*Distributed Virtual Hypercube Algorithm*
DNS	*Domain Name System*
Hi-ADSD	*Hierarquical Adaptive Distributed System-Level Diagnosis Algorithm*
HTTP	*Hyper Text Transfer Protocol*
HTTPS	*Secure Hyper Text Transfer Protocol*
I/O	*Input/Output* ou Entrada/Saída
TCP	*Transmission Control Protocol*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O objetivo do diagnóstico distribuído é permitir que os nodos sem-falha de um sistema determinem o estado – falho ou sem-falha – de todos os nodos do sistema. Assume-se que os nodos são capazes de testar outros nodos e os nodos sem-falha determinam o estado dos nodos testados corretamente. O algoritmo *Hierarchical Adaptive Distributed System-level Diagnosis (Hi-ADSD)* [9] é um algoritmo de diagnóstico distribuído que cria uma topologia virtual baseada em um hipercubo. O hipercubo é uma estrutura escalável por definição, apresentando características topológicas importantes como: simetria, diâmetro logarítmico e boas propriedades para tolerância a falhas. O algoritmo permite que todo nodo sem-falha de um sistema com N nodos determine o estado de todos os outros nodos com uma latência de no máximo $\log_2^2 N$ rodadas de teste. Entretanto, o número de testes executados no pior caso é quadrático. Este trabalho apresenta uma nova estratégia de testes para o algoritmo *Hi-ADSD*. Esta estratégia utiliza uma quantidade logarítmica de testes no pior caso. O algoritmo é adaptado para a nova estratégia de testes. Além disso, foi adotado o uso de *timestamps* para permitir que cada nodo obtenha informações de diagnóstico a partir de diversos outros nodos, consequentemente reduzindo a latência média. O novo algoritmo é especificado, suas provas formais são demonstradas e resultados experimentais obtidos por simulações são apresentados e comparados com o *Hi-ADSD*. A nova estratégia garante que no máximo $N \log_2 N$ testes são executados a cada $\log_2 N$ rodadas de teste. A latência máxima é mantida em $\log_2^2 N$ rodadas de teste. O novo algoritmo foi implementado como uma estratégia escalável de monitoramento e execução de experimentos integrado ao PlanetMon [23], um arcabouço para instalação, execução e monitação de experimentos no Planet-Lab. Experimentos foram realizados e mostraram que o desempenho do algoritmo em um ambiente realista condiz com as expectativas teóricas.

ABSTRACT

The goal of distributed diagnosis is to allow fault-free nodes of a system to determine the state – faulty or fault-free – of all nodes of the system. It is assumed that the nodes are capable of testing each other and that fault-free nodes can determine the state of tested nodes correctly. The *Hierarchical Adaptive Distributed System-level Diagnosis (Hi-ADSD)* [9] is a distributed diagnosis algorithm that creates a virtual topology based on a hypercube. A hypercube is a scalable structure by definition, presenting important topological features like: symmetry, logarithmic diameter and good fault tolerance properties. The algorithm allows every fault-free node of a system with N nodes to determine the state of every other node with a latency of at most $\log_2^2 N$ testing rounds. However, the number of executed tests in the worst case is quadratic. This work presents a new testing strategy for the *Hi-ADSD* algorithm. This strategy uses a logarithmic amount of tests in the worst case. The algorithm is adapted to the new testing strategy. Furthermore, the use of *timestamps* is adopted to allow each node to retrieve diagnosis information from several other nodes, thus reducing the average latency. The new algorithm is specified, formal proofs are given, and experimental results obtained by simulations are presented and compared with the *Hi-ADSD* algorithm. The new strategy ensures that at most $N \log_2 N$ tests are executed at each $\log_2 N$ testing rounds. The maximum latency is mantained in $\log_2^2 N$ testing rounds. The new algorithm was implemented as a scalable strategy to monitor and execute experiments integrated to PlanetMon [23], a framework to install, execute and monitor experiments in Planet-Lab. Experiments were executed and they have shown that the algorithm performance in a realistic environment is consistent with the teoretical limits.

CAPÍTULO 1

INTRODUÇÃO

Considere um sistema composto de N nodos, cada um destes nodos pode assumir um de dois estados: falho ou sem-falha. Um algoritmo de diagnóstico distribuído em nível de sistema [11, 20] permite que os nodos sem-falha do sistema determinem o estado de todos os outros nodos. Nodos desse sistema devem ser capazes de executar testes em outros nodos, podendo assim determinar em qual estado o nodo testado está. Assume-se que um testador sem-falha consegue determinar corretamente o estado do nodo testado.

Esta asserção referente à capacidade de um nodo sem-falha realizar testes com precisão faz parte do primeiro modelo proposto na área de diagnóstico em nível de sistema, apresentado por Preparata, Metze e Chien [21], chamado modelo PMC. Segundo o modelo, o diagnóstico clássico consiste na escolha de um conjunto de testes a serem realizados e na obtenção dos resultados destes testes para o diagnóstico do sistema.

O diagnóstico adaptativo [14], por sua vez, sugere que uma unidade determine quais testes vai executar baseando-se nos resultados dos testes executados previamente. Desta forma, cada unidade adapta o conjunto de testes que irá realizar de acordo com o resultado de testes anteriores. No diagnóstico adaptativo uma rodada de testes é definida como o tempo necessário para que as unidades sem-falha do sistema executem seus testes assinalados.

O modelo PMC considera a existência de um observador central, que é livre de falhas, para a realização do diagnóstico do sistema. Unidades realizam testes e enviam seus resultados a este observador, que os analisa e conclui qual o estado de todas as unidades do sistema. O diagnóstico distribuído em nível de sistema [15] é uma abordagem onde este observador central não existe, e cada unidade é responsável por realizar o diagnóstico do sistema como um todo.

Para que seja possível comparar o desempenho de diferentes algoritmos de diagnóstico são utilizadas duas métricas: latência e quantidade de testes. Em um sistema diagnosticável a ocorrência de um evento em uma unidade leva um certo tempo para ser detectada pelas outras unidades. 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. Define-se um evento como sendo a troca de estado de uma unidade do sistema, tanto de falho para sem-falha quanto de sem-falha para falho. Outra métrica importante para a avaliação de desempenho de estratégias de diagnóstico é a quantidade máxima de testes que o sistema realiza em uma rodada de testes.

O algoritmo *Adaptive Distributed System-level Diagnosis* (*Adaptive-DSD*) [4] foi o primeiro algoritmo de diagnóstico ao mesmo tempo adaptativo e distribuído. O algoritmo utiliza uma topologia em formato de anel para realizar testes. Todos os nodos do sistema tem identificadores sequenciais, de 0 a $N-1$. Um nodo é responsável por testar o nodo seguinte na topologia. Neste algoritmo uma rodada de testes é definida como o tempo necessário para que todos os nodos sem-falha do sistema testem pelo menos um nodo como sem-falha. Sua estratégia de testes se baseia no resultado dos testes anteriores. Caso o testador encontre um nodo falho, ele continuará fazendo testes em sequência até que encontre um nodo sem-falha, ou teste todos os nodos falhos. Este algoritmo possui latência de N rodadas de teste, enquanto a cada rodada de teste N testes são executados no máximo.

Em [9] o algoritmo *Hierarchical Adaptive Distributed System-Level Diagnosis (Hi-ADSD)* é introduzido. Este algoritmo apresenta uma latência de diagnóstico de no máximo $\log_2^2 N$ rodadas de teste para um sistema de N nodos. Os nodos são organizados em *clusters* progressivamente maiores. Os testes são executados de uma maneira hierárquica, inicialmente no *cluster* com dois nodos, indo para o *cluster* com quatro nodos, e assim sucessivamente, até que o *cluster* com $N/2$ nodos seja testado. Para que o nodo receba informações sobre um dado *cluster*, o nodo executa testes até que ele encontre um nodo não falho ou teste todos os nodos do *cluster* falhos. Considerando a rede como um todo, o maior número de testes executados em uma rodada de testes é $N^2/4$. Este cenário acontece quando $N/2$ nodos de um mesmo *cluster* estão falhos, e os outros $N/2$ nodos testam este *cluster* na mesma rodada de testes.

Diversos outros algoritmos de diagnóstico distribuído hierárquico já foram propostos, incluindo o algoritmo *Hi-ADSD with Detours* [7]. Neste algoritmo quando um nodo testa um nodo falho em um determinado *cluster*, o testador tenta obter informações sobre aquele *cluster* a partir de nodos fora deste *cluster*, ao invés de executar mais testes. Nesse algoritmo, esses caminhos, ditos *detours*, possuem tamanho máximo idêntico à distância no algoritmo original. Utilizando esses caminhos o algoritmo consegue reduzir o número de testes e a latência média observada no algoritmo *Hi-ADSD* original. Através de simulações e experimentos é possível verificar que o número de testes realizados a cada rodada de testes é logarítmico, porém até o momento tal fato não foi provado.

Outro algoritmo hierárquico, o algoritmo *Hi-ADSD with Timestamps* [8] introduz uma nova ideia ao algoritmo *Hi-ADSD with Detours*. Quando um nodo realiza um teste em um nodo sem-falha no algoritmo *Hi-ADSD with Detours*, este obtém informações sobre um conjunto de $N/2$ nodos. Assim é possível que um nodo obtenha informações sobre um outro nodo através de vários caminhos, sendo necessária uma estratégia para a detecção de qual destas informações é a mais recente. Para resolver este problema são criados os *timestamps*, que são contadores de mudança de estados, ou seja, a cada mudança de falho para sem-falha ou sem-falha para falho o contador referente ao nodo em questão é incrementado. Assim, analisando as informações recebidas sobre um nodo, é possível determinar qual informação é mais recente através da comparação de seus *timestamps*. Como os *timestamps* não influenciam na estratégia de testes do algoritmo original, este algoritmo mantém os piores casos tanto da latência quanto do número de testes executados pelos algoritmos anteriores. Experimentos mostram que a latência média diminui consideravelmente ao se utilizar *timestamps* [8].

No algoritmo *DiVHA (Distributed Virtual Hypercube Algorithm)* [5] cada nodo mantém localmente um grafo representando o sistema. Este grafo tem a mesma estrutura do hipercubo criado pelo algoritmo *Hi-ADSD*, inclusive utilizando as mesmas funções de assinalamento de testes. O algoritmo tenta manter as propriedades do hipercubo mesmo quando existem nodos falhos no sistema. Para isso, ele faz buscas e comparações no grafo local e determina quais os testes que cada nodo do sistema deve executar de acordo com a situação atual. *Timestamps* também são empregados neste algoritmo. É provado que o diâmetro do grafo de testes gerado pelo algoritmo é logarítmico. Além disso, mostra-se que a quantidade máxima de arestas é $N \log_2 N$. Sua maior desvantagem é o custo, pois em diversos momentos é necessário executar buscas nos grafos locais armazenados.

Neste trabalho apresentamos uma nova estratégia de testes para o algoritmo *Hi-ADSD*. Esta estratégia requer menos testes por rodadas de teste no pior caso, e mantém a latência de $\log_2^2 N$ rodadas. Na estratégia original utilizada pelo *Hi-ADSD* para determinar seus testes, um nodo pode ser testado por vários outros nodos. A ideia principal da nova estratégia de testes é evitar que estes testes extras sejam executados. Para isso, uma ordem de testes é estabelecida para cada nodo em cada *cluster*. Assim, para um nodo i fazer um teste no nodo j na rodada de testes s, o nodo i deve ser o primeiro nodo sem-falha na lista de nodos testadores de j na rodada s. Desta maneira conseguimos evitar múltiplos testes em um mesmo nodo. Como cada nodo recebe no máximo um teste por rodada de testes, a nova estratégia garante um número de testes executados no sistema de no máximo $N \log_2 N$ a cada $\log_2 N$ rodadas de teste, fato que é provado no trabalho. Resultados de simulação são apresentados comparando o algoritmo *Hi-ADSD* original com a nova estratégia, tanto em termos de latência como do número de testes executados.

O algoritmo com a nova estratégia de testes foi implementado como uma alternativa escalável para criar um *overlay* integrado ao PlanetMon [23], um arcabouço para execução e monitoração de experimentos no Planet-Lab [2]. Experimentos foram realizados para analisar o desempenho do algoritmo nestas condições.

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 apresenta conceitos e trabalhos relacionados. O Capítulo 3 descreve a nova estratégia de testes. O Capítulo 4 detalha uma implementação da nova estratégia de testes em um ambiente real. Por fim, o Capítulo 5 conclui o trabalho.

CAPÍTULO 2

DIAGNÓSTICO DISTRIBUÍDO HIERÁRQUICO

Considere um sistema S que consiste em um conjunto de N nodos, $n_0, n_1, ..., n_{N-1}$, alternativamente o nodo n_i pode ser chamado de nodo i. Assumimos que o sistema é totalmente conectado, ou seja, existe um canal de comunicação entre qualquer par de nodos e estes canais nunca falham. Cada nodo n_i pode estar falho ou sem-falha. Os nodos do sistema são capazes de testar outros nodos e determinar seu estado corretamente. O objetivo do diagnóstico é determinar o estado de todos os nodos do sistema. Neste trabalho assumimos que falhas são apenas do tipo *crash*, i.e. nodos falhos não interagem com outros nodos do sistema. Este capítulo apresenta conceitos e algoritmos de diagnóstico, focando em particular nos algoritmos hierárquicos.

2.1 O Modelo PMC

O primeiro trabalho de diagnóstico em nível de sistema foi apresentado pelos autores Preparata, Metze e Chien [21]. Naquele trabalho, um sistema é definido como um conjunto de unidades, onde uma unidade é uma porção do sistema não decomponível para propósitos de diagnóstico. As unidades possuem capacidade de testar outras unidades. Um assinalamento de testes é feito para determinar o conjunto de testes a serem executados no sistema. Assume-se que os enlaces de comunicação nunca falham, qualquer unidade pode testar outra unidade, ou seja, o sistema é representável por um grafo completo e nenhuma unidade testa a si mesma.

As unidades podem estar nos estados falho e sem-falha e toda unidade sem-falha é capaz de realizar testes com perfeição, enquanto que uma unidade falha produz resultados de testes arbitrários, quaisquer que sejam as condições das unidades testadas. O modelo prevê um tipo arbitrário de falha, desde que passível de detecção pelas unidades do sistema. Um teste envolve a aplicação de um conjunto de estímulos e a coleta das respostas correspondentes. O conjunto de resultados de todos os testes executados no sistema é chamado de síndrome. Assume-se que a síndrome do sistema é submetida a um observador central, responsável por decodificá-la e obter o diagnóstico.

Além disso, o modelo PMC também considera um sistema em termos de sua *diagnosability*, que traduzimos como diagnosticabilidade. Um sistema composto por N unidades é definido como t-diagnosticável se todas as suas unidades falhas podem ser identificadas, sendo t o número máximo de unidades falhas. Prova-se que em um sistema t-diagnosticável, N deve ser maior ou igual a $2t + 1$ e cada unidade deve ser testada por pelo menos t outras unidades, desde que duas unidades não se testem mutuamente [13]. Um sistema t-diagnosticável é definido como ótimo se $N = 2t + 1$ e se cada unidade é testada por exatamente t unidades. Na eventualidade de um número maior do que t, possivelmente todas as unidades do sistema, se tornem falhas, o algoritmo de diagnóstico deve recobrar o diagnóstico correto tão logo a medida de diagnosticabilidade, t, seja restaurada.

Conforme mencionado acima, o modelo PMC assume que o teste feito por uma unidade sem-falha é sempre correto, enquanto que o teste feito por uma unidade falha pode ou não ser correto, independente do estado da unidade testada. Esta asserção é conhecida como o modelo de invalidação simétrico de testes (*symmetric invalidation*), enquanto que outro modelo, o de invalidação assimétrica de testes (*asymmetric invalidation*), proposto em [3], caracteriza que uma unidade falha não pode avaliar outra unidade falha como sem-falha.

2.2 Diagnóstico Adaptativo

No diagnóstico clássico, tendo-se escolhido um conjunto fixo de testes e respeitando-se o limite de diagnosticabilidade, obtém-se a síndrome do sistema e identifica-se as unidades falhas. Hakimi e Nakajima [14] sugerem que, mantendo-se o limite de unidades falhas mencionado acima, o conjunto de testes seja escolhido de forma adaptativa com base nos resultados já obtidos, até encontrar no mínimo uma unidade sem-falha. A partir desta unidade se fazem testes para diagnosticar as demais unidades.

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

Generalizando, o diagnóstico adaptativo permite que os testes a serem executados sejam definidos com base nos resultados de testes executados anteriormente. Um conceito que facilita o diagnóstico adaptativo é o de rodadas de testes. Em uma rodada de testes todos os nodos sem-falha executam todos os seus testes assinalados. Para determinar os testes de uma rodada são utilizados resultados das rodadas anteriores.

2.3 Diagnóstico Distribuído

Paralelamente à abordagem adaptativa sugerida em [14], e devido aos inconvenientes de se ter um observador central que faça o diagnóstico, foi proposta a abordagem distribuída para o diagnóstico. Nesta abordagem os nodos constituintes de um sistema computacional distribuído tentam diagnosticar independentemente as condições das unidades do sistema. A informação usada por um nodo para produzir seu diagnóstico é o resultado de testes que ele realiza sobre nodos vizinhos juntamente com informações transmitidas com relação a resultados de testes realizados por outros nodos do sistema. Além disso, a asserção de que nodos sem-falha são capazes de realizar testes em outros nodos e determinar seu estado com precisão também é utilizada.

Hosseini, Kuhl e Reddy [15] introduzem um modelo de diagnóstico distribuído e apresentam o algoritmo *New-SELF*. Neste algoritmo também são consideradas falhas nos canais de comunicação, porém assume-se o uso de códigos de detecção de erro que tornariam evidentes as falhas em enlace. Como o envio de mensagens de diagnóstico pelo sistema é também sujeito a falhas, isso é tratado pelo algoritmo assumindo que um nodo que detecta outro nodo como falho deixa de aceitar mensagens de diagnóstico através daquele nodo. Assume-se, ainda, ambiguidade de falhas ao se detectar dificuldade de comunicação com nodos vizinhos: neste casos, não é possível saber se é o nodo vizinho ou o enlace de comunicação que está falho. Entretando, enlaces de comunicação falhos entre um par de nodos sem-falha são corretamente detectados como tal. Não somente falhas são diagnosticáveis pelo algoritmo, mas também a recuperação ou retorno de unidades falhas ao sistema são permitidas, e a introdução de nodos ou enlaces completamente novos ao sistema é também possível.

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

Um nodo somente aceita informações de diagnóstico de outros nodos que ele testa e diagnostica como sem-falha. O algoritmo é executado *on-line*, no sentido em que nodos podem falhar e serem reparados a qualquer momento, com a restrição de que um nodo não pode falhar e recuperar de forma não detectável durante o intervalo entre dois testes consecutivos por outro nodo. À semelhança do algoritmo de [14], o algoritmo *New-SELF* converge para o diagnóstico correto desde que menos do que t nodos estejam sem-falha, onde $t \leq (N-1)/2$.

2.4 O Algoritmo *Adaptive-DSD*

O algoritmo *Adaptive Distributed System-level Diagnosis (Adaptive-DSD)* [9] é um algoritmo de diagnóstico em nível de sistema ao mesmo tempo distribuído e adaptativo. Por ser distribuído, não existe uma unidade central para realização do diagnóstico. Por ser adaptativo, a configuração de testes varia conforme a configuração de falhas do sistema em um dado momento.

O algorimo prevê falhas e recuperações de nodos, mas o modelo não prevê falhas de enlaces e a topologia deve ser totalmente conectada. Como o algoritmo é adaptativo, o número permitido de nodos falhos é limitado a $N-1$, isto é, havendo apenas um nodo sem-falha, este é capaz de determinar o estado de todos os demais nodos do sistema e corretamente fazer o diagnóstico. Os algoritmos adaptivos possuem a noção de rodadas de testes, que consistem no período de tempo necessário para que todos os nodos sem-falha executem os testes que lhe foram atribuídos.

A estratégia de testes utilizada pelo algoritmo consiste em cada nodo testar o nodo seguinte, por exemplo, um nodo com identificador 2 testaria o nodo com identificador 3. Este cálculo é realizado utilizando aritimética em módulo N. Porém, caso o nodo testado seja diagnosticado como falho, os testes continuam até que um nodo sem-falha seja encontrado ou até que todos os nodos sejam testados como falhos. No caso do algoritmo *Adaptive-DSD*, uma rodada de testes se completa quando todos os nodos sem-falha executam testes até encontrar outro nodo sem-falha. Além disso, quando um nodo sem-falha é testado este repassa informações de diagnóstico sobre o estado do sistema. O algoritmo 1 mostra o pseudo-código do *Adaptive-DSD*.

A Figura 2.1 mostra um exemplo de execução do algoritmo *Adaptive-DSD* em um sistema com 6 nodos. É possível verificar o comportamento do algoritmo caso um testador encontre nodos falhos. O nodo 3 testa o nodo 4 como falho e adapta-se para testar o nodo 5, e assim por diante até que um nodo sem-falha, no caso o nodo 0, seja encontrado e suas informações de diagnóstico recuperadas.

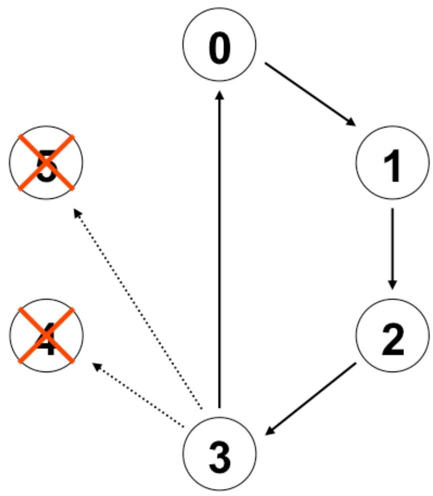

Figura 2.1: Algoritmo *Adaptive-DSD* executado em um sistema com 6 nodos.

Os autores provam que os testes executados pelos nodos sem-falha formam um ciclo após o algoritmo ter sido executado ao menos uma vez em cada nodo [9]. Após uma rodada de testes, existe um caminho de cada nodo sem-falha até qualquer outro nodo sem-falha. Este caminho forma um ciclo. Também é possível notar que após N rodadas de testes todos os nodos têm a mesma informação de diagnóstico.

O algoritmo *Adaptive-DSD* é ótimo em termos do número total de testes necessários, pois cada nodo é testado por no máximo um outro nodo. A latência de diagnóstico, entretanto, é de N rodadas de teste.

```
Algoritmo Adaptive-DSD executado no nodo i
repita
    node_to_test := i
    repita
        node_to_test := (node_to_test + 1) mod N
        teste(node_to_test)
        se node_to_test está sem-falha então
            obtém informações de diagnóstico
    até node_to_test está sem-falha ou todos os nodos estão falhos
    durma até o próximo intervalo de testes
para sempre
```

Algoritmo 1: Algoritmo *Adaptive-DSD*.

2.5 O Algoritmo *Hi-ADSD*

O algoritmo *Hierarchical Adaptive Distributed System-Level Diagnosis (Hi-ADSD)* [9] é hierárquico, além de ser distribuído e adaptativo, e realiza testes baseados em uma estratégia de dividir para conquistar. Esta hierarquia é baseada em um hipercubo. Quando todos os nodos do sistema estão sem-falha o grafo de testes é um hipercubo completo.

O hipercubo [16] é uma estrutura escalável por definição, apresentando características topológicas importantes como: simetria, diâmetro logarítmico e boas propriedades para tolerância a falhas. Em um hipercubo, um nodo i é vizinho de um nodo j se, e somente se, os identificadores de i e j têm apenas um *bit* diferente em suas representações binárias. Dizemos que um hipercubo com 2^d elementos tem dimensão d. Em um hipercubo de dimensão d, um caminho entre quaisquer dois nodos tem no máximo d arestas. A Figura 2.2 mostra um 3-hipercubo.

No *Hi-ADSD* os nodos são organizados em *clusters* para execução de testes e obtenção de informações de diagnóstico. O tamanho dos *clusters* é sempre uma potência de dois, e baseia-se na estrutura do hipercubo. No primeiro intervalo de testes um nodo testa outro nodo e vice-versa, formando um *cluster* de dois nodos. No segundo intervalo de testes

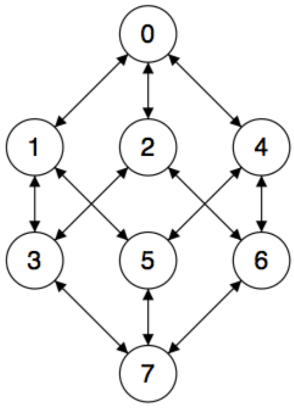

Figura 2.2: Um 3-hipercubo.

o testador testa um *"cluster"* de dois nodos, isto é, testa um nodo e obtém informação sobre o outro. No terceiro intervalo de testes o testador testa um cluster de quatro nodos, e assim sucessivamente até que no d-ésimo intervalo de testes o testador testa um *cluster* com $N/2$ nodos, e todos formam um *cluster* de N nodos, ou seja, o sistema todo. A Figura 2.3 mostra os *clusters* formados em um sistema com 8 nodos.

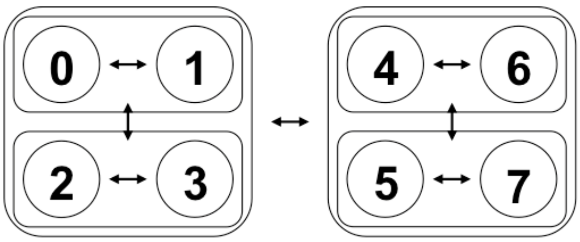

Figura 2.3: Clusters formados pelo algoritmo *Hi-ADSD* em um sistema de 8 nodos.

A lista ordenada de nodos testados por um determinado nodo i em um *cluster* de tamanho 2^{s-1} é dada pela função $c_{i,s}$. A Equação 2.1 mostra a especificação do algoritmo para esta função. Definimos o símbolo $\oplus$ como a operação *XOR*. Um exemplo de uso desta função em um sistema com 8 nodos é exposto na Tabela 2.1.

$$c_{i,s} = i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},s-1}, ..., c_{i\oplus 2^{s-1},1} \tag{2.1}$$

Quando um testador testa um nodo sem-falha, ele obtém informações de diagnóstico sobre os demais nodos do *cluster* do nodo testado. Quando o testador testa um nodo falho, ele continua testando nodos daquele *cluster* até encontrar um nodo não falho ou testar todos os nodos do *cluster* como falhos. A Figura 2.4 mostra os *clusters* testados

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	4,6,7,5	5,7,6,4	6,4,5,7	7,5,4,6	0,2,3,1	1,3,2,0	2,0,1,3	3,1,0,2

Tabela 2.1: $c_{i,s}$ para um sistema com 8 nodos.

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 a testar o *cluster* de tamanho 1, e assim sucessivamente.

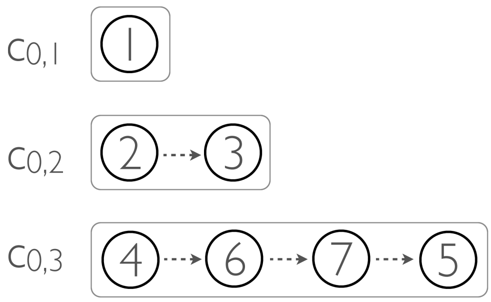

Figura 2.4: Clusters testados pelo nodo 0 executando o algoritmo *Hi-ADSD*.

O Algoritmo 2 mostra em pseudo-código o funcionamento do algoritmo *Hi-ADSD* executado no nodo i. O algoritmo realiza testes sequencialmente nos nodos retornados pela função $c_{i,s}$ até testar um nodo sem-falha. Ao testar um nodo sem-falha obtém-se informações de diagnóstico sobre os outros nodos daquele *cluster*. A latência do algoritmo *Hi-ADSD* é de no máximo $\log_2^2 N$ rodadas de teste.

Esta estratégia de testes pode gerar um número quadrático de testes. Por exemplo, suponha que um sistema de N nodos está com $N/2$ nodos falhos, e que tais nodos formam um *cluster* com $N/2$ nodos. Portanto, este é um sistema onde existe um *cluster* de $N/2$ nodos falhos e outro de $N/2$ nodos não falhos. Seguindo a estratégia acima, suponha que todos os nodos do *cluster* não falho estão na rodada de testes em que irão testar outro *cluster* de tamanho $N/2$. Assim, cada nodo não falho irá realizar $N/2$ testes, resultando em um total de $N/2 * N/2 = N^2/4$, ou seja, $O(N^2)$ testes em uma rodada de testes. Esta mesma complexidade pode ser obtida com um algoritmo de força bruta.

Algoritmo *Hi-ADSD* executado no nodo i
repita
  para $s \leftarrow 1$ ***até*** $\log_2 N$ **faça**
    repita
      node_to_test := próximo nodo em $c_{i,s}$
      teste(*node_to_test*)
      se *node_to_test está sem-falha* **então**
        atualiza informação de diagnóstico do *cluster*
    até *node_to_test está sem-falha* ***ou*** *todos os nodos em* $c_{i,s}$ *estão falhos*
    se *todos os nodos em* $c_{i,s}$ *estão falhos* **então**
      apaga informações de diagnóstico do *cluster*
    durma até o próximo intervalo de testes
para sempre

Algoritmo 2: Algoritmo *Hi-ADSD*.

2.6 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* [7]. 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 um nodo testa um nodo falho de um determinado *cluster* ele não continua a testar os demais nodos daquele *cluster*. Nesta situação, o *cluster* é dito como *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* (em português: desvios). Testes extras são realizados nos nodos do *cluster* bloqueado para os quais não foram encontrados *detours*.

Uma rodada de testes é definida como o período de tempo no qual todos os nodos sem-falha 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 um novo evento não ocorre no sistema até o evento anterior ter sido completamente diagnosticado.

O grafo de testes livres de falhas do sistema, $T(S)$, é o 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-falha no último intervalo de testes. Quando todos os nodos estão sem-falha $T(S)$ 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 2.5, 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 tais que $d_{i,p} \leq k$. Assim, de acordo com a Figura 2.5, $D_{0,1} = \{1, 2, 4\}$, $D_{0,2} = \{1, 2, 4, 3, 5, 6\}$ e $D_{0,3} = \{1, 2, 4, 3, 5, 6, 7\}$.

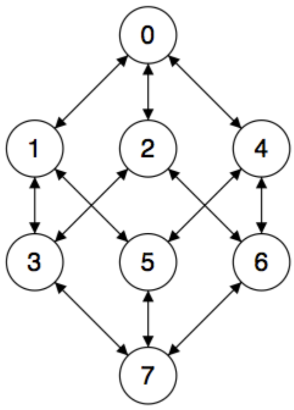

Figura 2.5: Grafo $T(S)$ para um sistema de 8 nodos.

$R_{i,s,p}$ é a lista ordenada de nodos que podem ser alcançados 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 equação abaixo. A Figura 2.5 mostra $R_{0,3,2} = \{6, 7\}$.

$$R_{i,s,p} = (c_{i,s} \cap D_{p,s-1}) - D_{i,d_{i,p}} \tag{2.2}$$

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 o nodo j pertence.

A Figura 2.6 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 pelo nodo 0 através do nodo 1, e as informações sobre os nodos 6 e 7 são obtidas através do nodo 2.

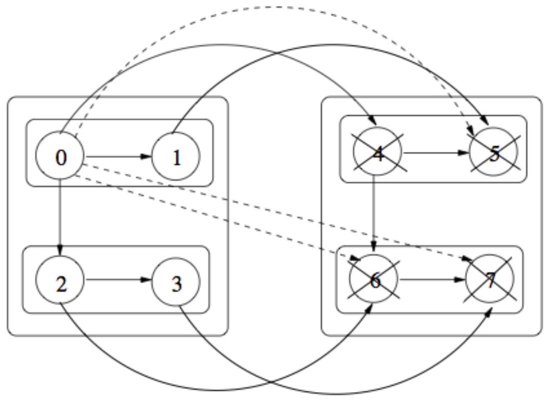

Figura 2.6: Sistema executando o algoritmo *Hi-ADSD with Detours*.

No algoritmo *Hi-ADSD with Detours* quando um nodo i testa um nodo $p \in c_{i,s}$ como falho, ao invés de continuar executando testes em $c_{i,s}$ o nodo i considera os nodos restantes em $c_{i,s}$ como *bloqueados*. Define-se como $B_{i,s}$ o conjunto de nodos bloqueados em $c_{i,s}$. Portanto, ao testar o nodo p como falho $B_{i,s}$ se torna o conjunto formado por $c_{i,s} - p$.

O Algoritmo 3 mostra o pseudo-código do algoritmo *Hi-ADSD with Detours* executado no nodo i. A especificação do algoritmo é feita utilizando as funções *more-info* e *more-tests*. Um testador i executa *more-info* depois de testar um nodo p sem-falha para determinar se é necessário obter informações adicionais através de p. A Equação 2.3 define a função *more-info*, onde i é o nodo testador e p é o nodo sem-falha testado ($p \in c_{i,s}$).

$$\text{more-info}(i, p) = R_{i,\log_2 N,p} \cap B_{i,s'},\ s' < s \leq \log_2 N \tag{2.3}$$

A função *more-tests* é executada quando um nodo falho é testado. Ela determina 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 equação 2.4, onde o nodo i é o testador e o nodo p é o nodo testado falho.

$$\text{more-tests}(i, s) = B_{i,s} - R_{i,s,p} \mid p \in c_{i,s'} \text{ e } 1 \leq s' < s \tag{2.4}$$

```
Algoritmo Hi-ADSD with Detours executado no nodo i
inicialize os conjuntos B_{i,s} como vazios
repita
    para s ← 1 até log_2 N faça
        node_to_test := primeiro nodo em c_{i,s}
        teste(node_to_test)
        se node_to_test está sem-falha então
            obter informações de diagnóstico de c_{i,s}
            atualizar B_{i,s}
            obter informações de diagnóstico de more-info(i,node_to_test)
            atualizar B_{i,s}
        senão
            B_{i,s} := c_{i,s} − node_to_test
            enquanto more-tests(i,s) ≠ ∅ faça
                k := primeiro nodo em more-tests(i,s)
                teste(k)
                se k está sem-falha então
                    obter informações de diagnóstico de B_{i,s}
                    atualizar B_{i,s}
                    obter informações de diagnóstico de more-info(i,k)
                    atualizar B_{i,s}

        durma até o próximo intervalo de testes
para sempre
```

Algoritmo 3: Algoritmo *Hi-ADSD with Detours*.

Simulações mostram que o *Hi-ADSD with Detours* 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 [7]. Os resultados mostram que o emprego de *detours* diminui o número de testes necessários para completar o diagnóstico do sistema.

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 with Detours*.

Apesar dos resultados dos experimentos e simulações do algoritmo mostrarem que em média o número de testes executados por rodada de testes é logarítmico, até o momento não existe uma prova formal para este fato.

2.7 O Algoritmo *Hi-ADSD with Timestamps*

Como nos demais algoritmos, no *Hi-ADSD with Timestamps* um evento ocorre quando um nodo falho se torna sem-falha ou quando um nodo sem-falha se torna falho. Os demais algoritmos hierárquicos baseados no modelo PMC assumem situações de falhas estáticas, não podendo haver um novo evento enquanto todos os nodos sem-falha 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* [8] 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. A utilidade do *timestamp* é permitir datar informações, distinguindo as informações mais atualizadas das menos atualizadas.

Ao testar um nodo sem-falha 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-falha são lidas informações de diagnóstico de todos os nodos que estão a uma distância de diagnóstico de até $\log_2 N - 1$ 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 do algoritmo *Hi-ADSD with Timestamps* é reduzida em relação ao algoritmo *Hi-ADSD with Detours*. O número máximo de testes é o mesmo do *Hi-ADSD with Detours* já que a estratégia de testes é a mesma. Simulações reportadas para sistemas de até 512 nodos mostram que o algoritmo *Hi-ADSD with Timestamps* é em média quatro vezes mais rápido para completar o diagnóstico de uma falha que o *Hi-ADSD with Detours* [8].

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 completar em todos os nodos sem-falha do sistema. 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-falha 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. 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-falha, ele atualiza suas informações 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 a informação é atualizada e 0 caso contrário.

Ao reparar-se de uma falha o nodo atribui a todos os seus *u-bits* o valor 0. Ao encontrar um nodo sem-falha 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-falha atualizado) então ele precisa atribuir todos os *u-bits* para 1. Durante a inicialização do algoritmo ocorre algo similar, pois apesar de encontrar nodos sem-falha estes nodos ainda não possuem informação de diagnóstico atualizadas, ou seja, têm o *u-bit* igual a 0.

O *timestamp* é um mecanismo que permite datar a informação de diagnóstico e é implementado como um contador de mudanças de estado. Sempre 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 o nodo j ele precisa comparar o *timestamp* do estado de j obtido com o seu *timestamp* local. Se o *timestamp* obtido é menor que o local então a informação de diagnóstico é desatualizada e i não deve atualizar o estado de j. Caso contrário, se o *timestamp* local de j for menor que o obtido, a informação é mais nova, então o *timestamp* local deve ser atualizado.

2.8 O Algoritmo *DiVHA*

O algoritmo *Distributed Virtual Hypercube Algorithm (DiVHA)* [5] tem como objetivo, anterior ao diagnóstico, determinar o conjunto de testes a ser executado pelos nodos do sistema dado seu estado. Para isso o algoritmo utiliza-se de um grafo local para realizar buscas e comparações, computando quais os testes cada nodo do sistema deve realizar. A construção do hipercubo através da função $c_{i,s}$ do algoritmo *Hi-ADSD* original e a ideia de *timestamps* do algoritmo *Hi-ADSD with Timestamps* foram ambas empregadas na construção do algoritmo *DiVHA*.

Seja o grafo de testes, $T(S)$, um grafo direcionado cujos vertices são os nodos de um sistema S. Existe uma aresta direcionada do nodo i para o nodo j se o nodo i testa o nodo j na rodada de testes mais recente. Assim $T(S)$ representa todos os testes executados na última rodada de testes. Quando todos os nodos estão no estado sem-falha, $T(S)$ é um hipercubo. Quando alguns nodos são considerados falhos, $T(S)$ deixa de ser um hipercubo e novas arestas são adicionadas com o objetivo de preservar duas de suas propriedades: (1) o diâmetro logarítmico, ou seja, a distância entre qualquer par de nodos em $T(S)$ nunca é maior que $\log_2 N$; e (2) o número total de arestas em $T(S)$ nunca é maior que $N \log_2 N$.

Para que tais propriedades sejam respeitadas, o algoritmo é executado de uma maneira incremental, ou seja, considerando os *clusters* do menor para o maior. Primeiramente o grafo de testes $T(S)$ contém os nodos do sistema e para cada nodo sem-falha i existem arestas de i para $c_{i,1}$. Assim a distância em $T(S)$ entre um nodo sem-falha i e todos os seus nodos em $c_{i,1}$ (que contém apenas um nodo) é 1. O processo é repetido para a distância 2, quando ao final de sua execução as distâncias em $T(S)$ entre um nodo sem-falha i e os nodos em $c_{i,2}$ devem ser menores ou iguais a 2, e assim por diante até que seja executada a iteração para distância $\log_2 N$. Sempre que não existe um caminho ou a distância entre dois nodos é maior do que a distância máxima permitida no momento, novas arestas são adicionadas. Após a execução do algoritmo, as distâncias entre um nodo sem-falha i e os nodos em $c_{i,s}$ são todas menores ou iguais a s.

Esta estratégia para adição de arestas permite formar um hipercubo quando todos os nodos estão no estado sem-falha. Caso contrário, quando existem nodos considerados falhos, as arestas adicionais são distribuídas entre os nodos sem-falha de forma balanceada.

O algoritmo *DiVHA* garante que a quantidade máxima de testes executados no sistema e a latência para detecção de eventos são logarítmicos. Porém sua execução pode ser custosa devido ao método de computação de arestas extras, visto que cada nodo calcula quais testes todos os nodos devem fazer de acordo com a configuração atual do sistema e para isso necessitam realizar diversas buscas no grafo que representa o sistema.

2.9 Outras Abordagens

Existem outras abordagens para realizar o diagnóstico em um sistema distribuído, por exemplo, o diagnóstico baseado em comparações [11]. Neste tipo de diagnóstico, ao invés de executar testes um testador (dito comparador) envia uma tarefa para ser executada por dois nodos do sistema. Após receber os resultados das execuções das tarefas estes são comparados. Se a comparação resultar em igualdade e se a unidade comparadora estiver sem-falha, então os dois nodos são considerados sem-falhas, caso contrário um ou ambos os nodos estão falhos. Um modelo importante de diagnóstico baseado em comparações é o modelo chamado de MM [19], que assume que as comparações são realizadas pelas próprias unidades do sistema, e apenas os resultados das comparações são enviados para o observador central que determina quais unidades estão falhas. Diversas outras abordagens baseadas em comparação existem [11].

Modelos de diagnóstico probabilísticos foram primeiramente abordados em [6], onde este método foi utilizado para diagnóstico em sistemas baseados em múltiplos processadores. Modelos probabilísticos assumem uma probabilidade de falha, isto é, a probabilidade de uma unidade produzir uma saída incorreta. A principal vantagem do modelo probabilístico é empregar, em geral, um menor número de testes em comparação com outras abordagens.

Outro algoritmo de diagnóstico é o *HeartbeatComplete* [25], que baseia-se em mensagens de *broadcast*. Ele assume uma rede síncrona, onde os atrasos nas comunicações são limitados. Além disso a rede entrega mensagens de maneira confiável e se mantém conectada todo o tempo. O algoritmo consiste em enviar mensagens de *heartbeat* através de *broadcast*. Como a rede é síncrona, os nodos da rede podem estabelecer um intervalo de tempo em que eles devem receber um *heartbeat* de outro nodo. Assim, caso este *heartbeat* não seja recebido no tempo estipulado, o nodo é considerado como falho. Quando o *heartbeat* é recebido, o nodo é assinalado como *working* e o temporizador para espera da próxima mensagem é reiniciado.

Ainda diversas outras abordagens já foram propostas para o diagnóstico em nível de sistema. Entre elas o diagnóstico de redes de topologia arbitrária [10], o diagnóstico aproximado baseado em técnicas de Inteligência Artificial [12], o diagnóstico condicional [17], o diagnóstico de falhas intermitentes e transientes [24], para citar alguns.

CAPÍTULO 3

O ALGORITMO DO HIPERCUBO VIRTUAL

Este capítulo descreve o algoritmo proposto por este trabalho, chamado de Hipercubo Virtual (*Virtual Hypercube*). O novo algoritmo representa uma nova estratégia de testes para o algoritmo *Hi-ADSD* que garante uma quantidade máxima de testes logarítmica mantendo o mesmo limite para a latência. O restante deste capítulo está organizado da forma a seguir. A Seção 3.1 especifica e descreve o novo algoritmo. A Seção 3.2 exibe uma nova definição para a função $c_{i,s}$ utilizada no novo algoritmo. Na Seção 3.3 são desenvolvidas as provas necessárias para mostrar que o número de testes é no máximo $N \log_2 N$ e que a latência é no máximo $\log_2^2 N$. Por fim, na Seção 3.4 resultados experimentais obtidos através de simulações são apresentados e comparados com o algoritmo *Hi-ADSD*.

3.1 O Algoritmo

Considere um sistema S que consiste em um conjunto de N nodos, $n_0, n_1, \ldots, n_{N-1}$. Alternativamente chamamos o nodo n_i de nodo i. Assumimos que o sistema é completamente conectado, ou seja, existe um canal de comunicação entre qualquer par de nodos. Cada nodo pode assumir um de dois estados, falho ou sem-falha. A combinação dos estados de todos os nodos do sistema constitui a situação de falha do mesmo. Os nodos realizam testes em outros nodos em intervalos de testes, e testes feitos por nodos sem-falha são sempre corretos.

No algoritmo *Virtual Hypercube* os nodos são agrupados em *clusters* para realização de testes. Os *clusters* são conjuntos de nodos cujo tamanho é sempre uma potência de 2. O sistema como um todo é um *cluster* de N nodos. A Figura 3.1 mostra um sistema com 8 nodos organizado em *clusters*.

O algoritmo *Virtual Hypercube* executado pelo nodo i é apresentado em pseudo código no Algoritmo 4. No primeiro intervalo de testes o nodo i realiza testes em nodos do *cluster*

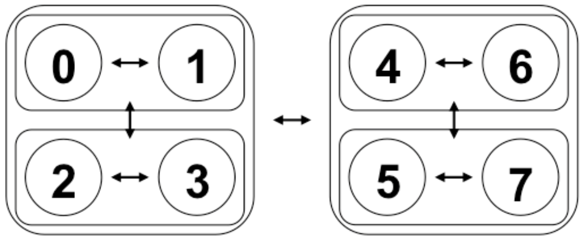

Figura 3.1: Um sistema de 8 nodos organizado em clusters.

que contém um nodo ($c_{i,1}$), no segundo intervalo de testes em nodos do *cluster* que contém dois nodos ($c_{i,2}$), e assim por diante, até que os nodos do *cluster* que contém $2^{\log_2 N - 1}$, ou $N/2$ nodos ($c_{i,\log_2 N}$). A equação 3.1 define a função $c_{i,s}$ para todo $i \in \{0, 1, \ldots, N-1\}$ e $s \in \{1, 2, \ldots, \log_2 N\}$. A função $c_{i,s}$ retorna a lista ordenada de nodos que o nodo i poderá testar durante a rodada de testes s. Define-se o símbolo $\oplus$ como a operação ou exclusivo (*XOR*). A Seção 3.2 descreve detalhadamente o funcionamento desta função.

$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i \oplus 2^{s-1},k} \mid k = 1, 2, \ldots, s-1 \tag{3.1}$$

O novo algoritmo é baseado na seguinte regra: antes do nodo i executar um teste no nodo $j \in c_{i,s}$, ele verifica se ele é o primeiro nodo sem-falha em uma lista ordenada de nodos que podem testar o nodo j na rodada s. Depois que o nodo i testa o nodo $j \in c_{i,s}$ como sem-falha, ele obtém novas informações de diagnóstico do nodo j sobre todos os nodos do sistema. Este processo é repetido para todos os nodos pertencentes a $c_{i,s}$, finalizando uma rodada de testes.

Como um testador pode obter informações sobre um determinado nodo através de vários outros nodos, o algoritmo utiliza *timestamps*. Inicialmente todos os nodos são considerados sem-falha e o *timestamp* correspondente é zero. Quando um evento é detectado, isto é, um nodo sem-falha se torna falho ou vice-versa, o *timestamp* correspondente é incrementado. Assim, um *timestamp* par corresponde a um nodo sem-falha e um *timestamp* ímpar corresponde a um nodo falho.

O uso de *timestamps* torna o algoritmo capaz de diagnosticar eventos dinâmicos, ou seja, eventos que ocorrem antes do evento anterior ter sido completamente diagnosticado. Considerando este modelo de falhas dinâmicas é necessário especificar um procedimento para a recuperação dos nodos que são reparados durante a execução do algoritmo. Utilizamos a mesma estratégia introduzida pelo algoritmo *Hi-ADSD with Timestamps*, onde um campo chamado *u-bit* é utilizado para representar se um valor de *timestamp* é válido ou não.

O nodo i mantém um vetor de *timestamps* $t_i[0..N-1]$. Após obter informação de um nodo j testado como sem-falha, o nodo i atualiza um *timestamp* local apenas quando o valor correspondente obtido for maior que o atual. Uma vantagem adicional desta abordagem é que a latência média do algoritmo é melhorada, pois testadores podem obter novas informações de diagnóstico sobre todos os nodos do sistema através de qualquer nodo testado.

Para evitar que o todo o vetor $t_j[]$ seja transferido quando o nodo i testa o nodo j como sem-falha, é possível implementar uma solução simples na qual o nodo j só envia novas informações, ou seja, informações que mudaram desde a última vez que o nodo j foi testado pelo nodo i.

É importante observar que o sistema é assíncrono, ou seja, em um certo momento, nodos diferentes do sistema podem estar testando *clusters* de tamanhos diferentes. Um nodo executando o algoritmo não é capaz de determinar quais testes estão sendo realizados por outros nodos do sistema em um dado momento. Mesmo se inicialmente os nodos estão sincronizados, após a falha e recuperação de alguns deles o sistema perderia sua sincronicidade. Este fato gera consequências no desempenho do algoritmo, que são tratadas na Seção 3.3.

3.2 A Função $c_{i,s}$

A função $c_{i,s}$ é a base do algoritmo, pois ela é capaz de determinar para um nodo i quais são os *clusters* que ele deverá testar durante cada rodada de teste. A função $c_{i,s}$ retorna a lista ordenada de nodos que o nodo i poderá testar durante a rodada de testes s. Definimos o símbolo $\oplus$ como a operação ou exclusivo (XOR). A equação 3.2 mostra a definição da função $c_{i,s}$, e um exemplo de uso da função para um sistema com 8 nodos pode ser visto na Tabela 3.1.

```
Algoritmo Virtual Hypercube executado no nodo i
repita
    para s ← 1 até log₂ N faça
        para todo j ∈ c_{i,s} | i é o primeiro nodo sem falha ∈ c_{j,s} faça
            teste(j)
            se j está sem-falha então
                se t_i[j] mod 2 = 1 então t_i[j] = t_i[j] + 1
                obtém informações de diagnóstico
            senão
                se t_i[j] mod 2 = 0 então t_i[j] = t_i[j] + 1
        durma até o próximo intervalo de testes
para sempre
```

Algoritmo 4: Algoritmo *Virtual Hypercube*.

$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i \oplus 2^{s-1},k} \mid k = 1, 2, \ldots, s-1 \tag{3.2}$$

Para entendermos melhor como a função $c_{i,s}$ funciona, primeiramente vamos lembrar que em um hipercubo existe uma aresta entre o nodo i e o nodo j se, e somente se, na representação binária de i e j existe apenas um *bit* diferente [16]. Também devemos lembrar que dado um número i podemos inverter um de seus bits, utilizando a operação ou exclusivo (*XOR*), simplesmente fazendo $i \oplus 2^x$, onde x é a posição do bit que gostariamos de inverter e $\oplus$ é a operação ou exclusivo (*XOR*).

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	5,4,7,6	6,7,4,5	7,6,5,4	0,1,2,3	1,0,3,2	2,3,0,1	3,2,1,0

Tabela 3.1: $c_{i,s}$ para um sistema com 8 nodos.

A função $c_{i,s}$ mostrada na equação 3.2 é uma contribuição se comparada à versão original especificada pelo algoritmo *Hi-ADSD*. A ordem criada por ela tem propriedades importantes para o novo algoritmo, vistas a seguir.

A Tabela 3.2 mostra um exemplo da função $c_{i,s}$ calculada para um sistema com 8 nodos onde $s = 3$. Podemos ver que nenhum nodo é repetido nas colunas da tabela, ou seja, $\forall x, y \in \{0, 1, ..., N-1\}, k \in \{0, ..., 3\} \mid x \neq y : c_{x,3}[k] \neq c_{y,3}[k]$. Para provarmos este fato, utilizamos indução.

i	$c_{i,3}[0]$	$c_{i,3}[1]$	$c_{i,3}[2]$	$c_{i,3}[3]$
0	4	5	6	7
1	5	4	7	6
2	6	7	4	5
3	7	6	5	4
4	0	1	2	3
5	1	0	3	2
6	2	3	0	1
7	3	2	1	0

Tabela 3.2: Detalhe da $c_{i,3}$ para um sistema com 8 nodos.

Teorema 1: $\forall i, j, s, p \mid i \neq j : c_{i,s}[p] \neq c_{j,s}[p]$.

Prova: A prova será por indução em s.

Base: Considere $s = 1$. Para um nodo i a função $c_{i,1} = i \oplus 2^{s-1} = i \oplus 1$. Como 1 em binário tem apenas o *bit* menos significativo como 1, a operação *XOR* irá inverter o *bit* menos significativo de i. Considere i como sendo par, ou seja, seu *bit* menos significativo é 0: assim temos que $i \oplus 1 = i + 1$. Considere i como sendo ímpar, ou seja, seu *bit* menos significativo é 1: assim temos que $i \oplus 1 = i - 1$. Portanto, para qualquer $i, j \mid i \neq j$, $c_{i,1}[0] \neq c_{j,1}[0]$.

Hipótese de indução: Para todo $i, j, p \mid i \neq j : c_{i,k}[p] \neq c_{j,k}[p]$.

Passo: Considere $s = k + 1$.

Para um nodo i a função $c_{i,k+1} = i \oplus 2^k \parallel c_{i \oplus 2^k,1} \parallel c_{i \oplus 2^k,2} \parallel ... \parallel c_{i \oplus 2^k,k}$. Considere a operação $i \oplus 2^k$. Como 2^k tem apenas o k-ésimo *bit* como 1, a operação *XOR* irá inverter o k-ésimo *bit* de i. Suponha que $j = i \oplus 2^k$, temos que $c_{j,k+1} = (i \oplus 2^k) \oplus 2^k, ...$ Temos então que $(i \oplus 2^k) \oplus 2^k = i$, ou seja, se $c_{i,k+1}[0] = j$ então $c_{j,k+1}[0] = i$.

Como $c_{i,k+1}[0]$ é único para cada i e pela hipótese $c_{i \oplus 2^k,1}$, $c_{i \oplus 2^k,2}$, ... , $c_{i \oplus 2^k,k}$ também são únicos, $c_{i,k+1}$ é único, e portanto, $c_{i,k+1}[p] \neq c_{j,k+1}[p], \forall j \neq i, p$.

$\square$

Além da função $c_{i,s}$ representar quais nodos o nodo i pode testar na rodada de testes s, ela também representa para o nodo i durante a rodada de testes s, a ordem dos nodos que poderão testar o nodo i nesta rodada. Tecnicamente qualquer ordem pode ser utilizada para representar os nodos que podem testar um outro nodo em uma rodada de testes.

Entretanto uma função que satisfaça o *Teorema 1* garante que a cada rodada de testes todos os nodos sem-falha irão realizar pelo menos um teste.

Considere, por exemplo, $c_{5,3} = \{1, 0, 3, 2\}$. Na terceira rodada de testes o nodo 5 irá verificar se ele é o primeiro nodo sem-falha nas listas ordenadas $c_{1,3}$, $c_{0,3}$, $c_{3,3}$ e $c_{2,3}$, respectivamente, e realizar o teste em caso positivo. É garantido que o nodo 5 irá realizar pelo menos um destes testes pois em alguma destas ordens ele será o primeiro. Por outro lado, os nodos 1, 0, 3 e 2, também na terceira rodada de testes, estando todos sem-falhas, irão verificar qual o primeiro nodo sem-falha em $c_{5,3}$, fazendo com que apenas um deles teste o nodo 5.

O uso de uma função $c_{i,s}$ que satisfaz o *Teorema 1* permite, de uma certa forma, manter as propriedades logarítmicas mesmo quando nodos se tornam falhos. A nova função $c_{i,s}$ mantém uma ordem progressiva de acordo com o hipercubo, fazendo com que as arestas criadas por nodos falhos conectem os nodos sem-falha de uma maneira mais similar ao hipercubo original. Suponha que em uma função $c_{i,s}$ o valor de $c_{i,3}$ fosse igual a $\{4, 5, 6, 7\}$ para todo $i \in \{0, 1, 2, 3\}$, ou seja, caso o nodo 4 esteja sem-falha ele será responsável por testar os nodos 0, 1, 2 e 3. Dependendo da função $c_{i,s}$ a carga de testes sobre um nodo pode chegar a ordem de N, e poucas falhas podem fazer com que o desempenho do algoritmo seja prejudicado consideravelmente. A função $c_{i,s}$ proposta distribui a responsabilidade pelos testes de forma igualitária entre todos os nodos.

3.3 Provas Formais

Nesta seção são apresentadas as provas de corretude e pior caso da latência de diagnóstico do algoritmo. Para estas provas assumimos que a situação de falhas do sistema não muda por um período suficiente de tempo. Vale lembrar que as provas de corretude dos algoritmos *Hi-ADSD* e *Adaptive-DSD* também consideram esta asserção.

Definição 1: *O grafo $T(S)$ é um grafo direcionado cujos vértices são os nodos de S. Para cada nodo i, e para cada cluster $c_{i,s}$, existe uma aresta de i para j se i testou j*

como sem-falha na rodada de testes mais recente em que i testou o cluster $c_{i,s}$.

Lema 1: *Para qualquer nodo i, qualquer s, e a qualquer instante de tempo t_i, são necessárias, no máximo, $\log_2 N$ rodadas de testes para que o nodo i teste o cluster $c_{i,s}$.*

Prova: O lema segue da definição do algoritmo, ou seja, em um determinado intervalo de testes o nodo i testa pelo menos um nodo de $c_{i,s}$. Como o nodo i testa $\log_2 N$ *clusters*, são necessárias $\log_2 N$ rodadas de testes para que todos os *clusters* sejam testados. Portanto, no pior caso, para t_i imediatamente após o teste de um dado *cluster*, poderá levar até $\log_2 N$ rodadas de testes para que este *cluster* seja testando novamente.

$\square$

Teorema 2: *O menor caminho entre qualquer par de nodos sem-falha em $T(S)$ contém, no máximo, $\log_2 N$ arestas.*

Prova: A prova será por indução em t, para um sistema de 2^t nodos.

Base: Considere um sistema de 2^1 nodos. Cada nodo testa o outro, portanto o menor caminho entre eles em $T(S)$ contém uma aresta.

Hipótese de indução: Assuma que para um sistema com 2^k nodos o menor caminho entre qualquer par de nodos em $T(S)$ contém, no máximo, k arestas.

Passo: Por definição, um sistema de 2^{k+1} nodos é composto por dois *clusters* de 2^k nodos. Considere o subgrafo de $T(S)$ que contém apenas os nodos de um destes *clusters*. Pela hipótese anterior o menor caminho entre qualquer par de nodos neste subgrafo contém, no máximo, k arestas. Considere quaisquer dois nodos i e j. Se i e j estão no mesmo *cluster* de 2^k nodos, o menor caminho entre eles em $T(S)$ contém, no máximo, k arestas. Agora considere o caso em que i e j estão em *clusters* de tamanho 2^k diferentes. Sem perda de generalidade, considere o menor caminho entre i e j. O nodo j recebe um teste de algum nodo do *cluster* ao qual o nodo i pertence: chame este nodo de p. Em $T(S)$, a menor distância entre p e j é de uma aresta, e a menor distância de i até p é, no máximo, k arestas. Portanto, o menor caminho de i a j contém, no máximo, $k + 1$ arestas.

$\square$

Teorema 3: *Considere uma situação de falha em um dado momento. Após, no máximo,* $\log_2^2 N$ *rodadas de testes, cada nodo que permaneceu sem-falha por este período corretamente determina aquela situação de falha.*

Prova: Foi provado no Teorema 2 que o menor caminho entre qualquer par de nodos em $T(S)$ tem, no máximo, $\log_2 N$ arestas. Segundo o Lema 1, cada um dos testes correspondentes a uma aresta em $T(S)$ pode levar até $\log_2 N$ rodadas de testes para ser executado no pior caso. Então, temos até $\log_2 N$ testes diferentes para serem executados, e cada um deles pode levar até $\log_2 N$ rodadas de testes para ser executado. Então, no total, estes testes podem levar no máximo $\log_2 N * \log_2 N$ rodadas de testes para serem executados. Portanto, podem ser necessárias até $\log_2^2 N$ rodadas de teste para um nodo sem-falha obter informação de diagnóstico sobre um evento em S.

□

Teorema 4: *O número de testes executados por todos os nodos do sistema executando o algoritmo Virtual Hypercube é no máximo* $N \log_2 N$ *a cada* $\log_2 N$ *rodadas de testes.*

Prova: Considere os testes executados no nodo j. Para cada $c_{j,s} \mid s = 1, ..., \log_2 N$, o nodo j é testado pelo primeiro nodo sem-falha em $c_{j,s}$. Cada nodo sem-falha em $c_{j,s}$ utiliza a mesma lista ordenada de nodos que contém ele mesmo para verificar se ele é o primeiro nodo sem-falha ou não. Em caso afirmativo ele deve testar o nodo j. Assim cada nodo é testado no máximo $\log_2 N$ vezes a cada $\log_2 N$ rodadas de teste. Como o sistema tem N nodos, o número total de testes é no máximo $N \log_2 N$.

□

Deve ficar claro que, assim como nos algoritmos *Hi-ADSD* e *Adaptive-DSD*, o limite do número de nodos falhos para que os nodos sem-falha realizem um diagnóstico consistente é $N - 1$. Neste caso, se $N - 1$ nodos estão falhos, o nodo sem-falha irá testar todos os nodos para diagnosticar o sistema.

Também devemos notar que o número de nodos do sistema, N, não necessariamente precisa ser uma potência de 2. Neste caso, os nodos testadores devem ignorar os $2^{\lceil \log_2 N \rceil} - N$ nodos inexistentes durante testes e recuperação de informações de diagnóstico.

3.4 Resultados Experimentais

Os resultados experimentais do algoritmo *Virtual Hypercube* foram obtidos tanto através de simulação, apresentados nesta seção, como em execução no Planet-Lab, apresentados no próximo capítulo. As simulações foram conduzidas utilizando a linguagem de simulação baseada em eventos discretos, SMPL [18]. Nodos foram modelados como *facilities* SMPL e cada nodo foi identificado por um número de *token* SMPL. As falhas foram modeladas como um nodo sendo reservado, que é apenas um estado interno da ferramenta de simulação. Durante cada teste, o estado do nodo é verificado, e se este está reservado ele é considerado falho. Caso contrário o nodo é considerado sem-falha. Cada nodo inicia seus testes em um *cluster* escolhido aleatoriamente, fazendo com que os nodos do sistema não estejam completamente sincronizados.

Todos os experimentos foram executados em uma rede de 512 nodos. Os resultados mostrados são a latência média e a quantidade de testes média. A latência é o número de rodadas de testes necessárias para que todos os nodos sem-falha detectem um evento. A quantidade de testes mostrada representa o número de testes realizados em média no sistema em uma situação estável, ou seja, a quantidade de testes após todos os nodos diagnosticarem o evento em questão durante $\log_2 N$ rodadas de testes. A simulação leva em conta a asserção de que um evento ocorre apenas quando o evento anterior tiver sido completamente diagnosticado pelo sistema.

As Figuras 3.2 e 3.3 mostram a quantidade de testes e a latência média, respectivamente, em uma rede onde os nodos falham de uma maneira a simular o pior caso do algoritmo *Hi-ADSD*. Os nodos de identificadores 0 a 255 falham de maneira aleatória até que todos estejam falhos e em seguida o mesmo acontece com os nodos de identificadores 256 a 511. Assim, com $N/2$ nodos falhos no mesmo *cluster*, comparamos o comportamento do *Virtual Hypercube* com o *Hi-ADSD*.

Pode-se observar que o *Hi-ADSD* tem um comportamento quadrático quando o número de nodos falhos se aproxima de $N/2$, chegando ao seu pior caso quando o número de testes chega a $N^2/4 = 65536$. Durante os mesmos casos de testes, o algoritmo *Virtual Hypercube* se comporta de maneira logarítmica, nunca passando de seu limite de $N \log_2 N = 4608$

testes por $\log_2 N$ rodadas de testes. Analisando a latência média que ambos algoritmos apresentaram, podemos ver que a nova estratégia de testes apresenta uma diferença considerável se comparada à estratégia original, apesar de ambas apresentarem uma latência máxima teórica de $\log_2^2 N$ rodadas de testes. Esta diferença se deve ao fato de que o algoritmo *Hi-ADSD* original não utiliza *timestamps*, que diminuem consideravelmente a latência.

As Figuras 3.4 e 3.5 mostram a quantidade de testes e a latência média, respectivamente, em uma rede onde os nodos falham de uma maneira aleatória. Novamente podemos ver que a latência média do *Virtual Hypercube* é mais baixa que a do *Hi-ADSD*. Um resultado a ser destacado é que, na média, o número de testes executados pelos dois algoritmos é muito parecido. Estes resultados foram obtidos a partir da execução de cerca de 7000 simulações de cenários de falhas para os dois algoritmos, porém existem 511! cenários de falhas possíveis. Como os nodos que irão falhar são escolhidos aleatoriamente, um caso como o pior caso do algoritmo *Hi-ADSD* tem uma probabilidade muito pequena de acontecer.

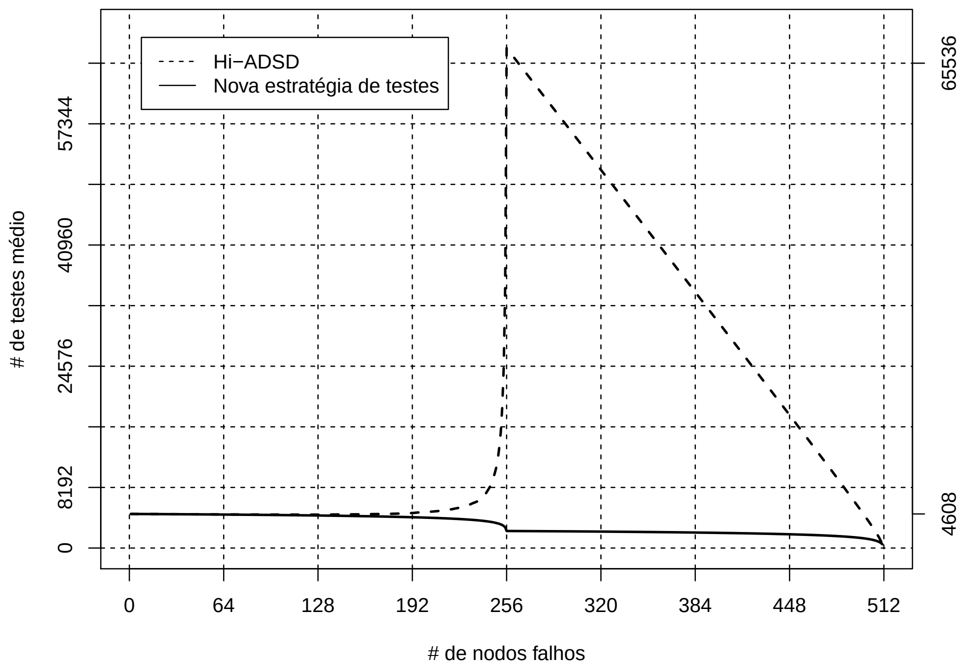

Figura 3.2: Número de testes médio em uma rede estável de 512 nodos no pior caso do algoritmo *Hi-ADSD*.

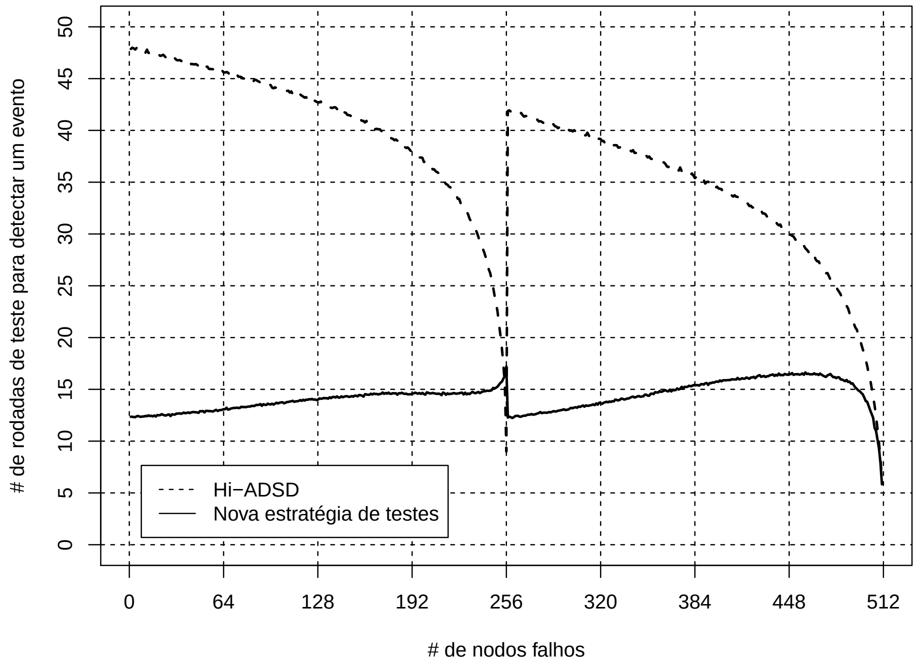

Figura 3.3: Latência média para detecção de um evento em uma rede de 512 nodos no pior caso do algoritmo *Hi-ADSD*.

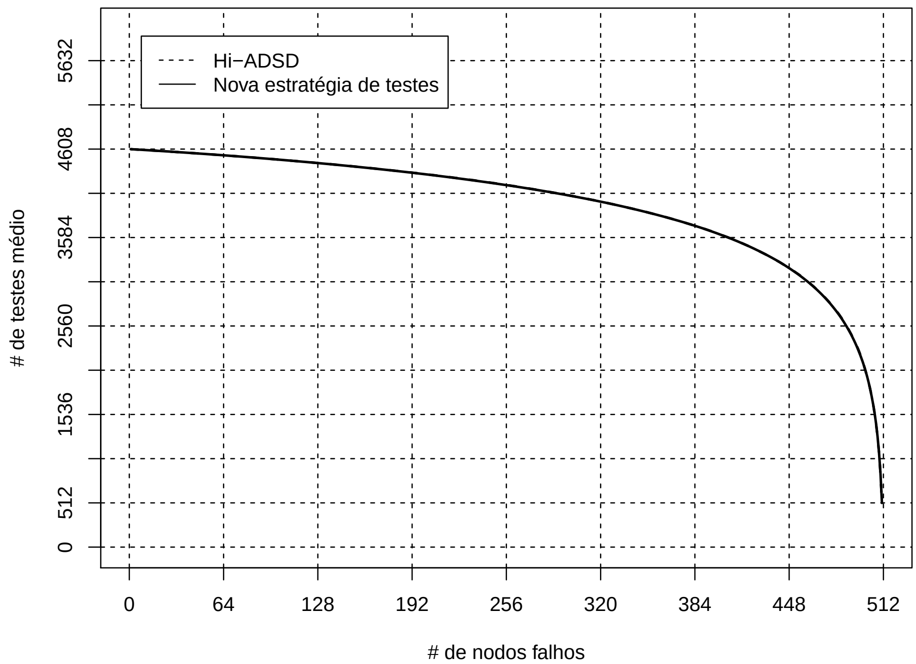

Figura 3.4: Número de testes médio em uma rede estável de 512 nodos.

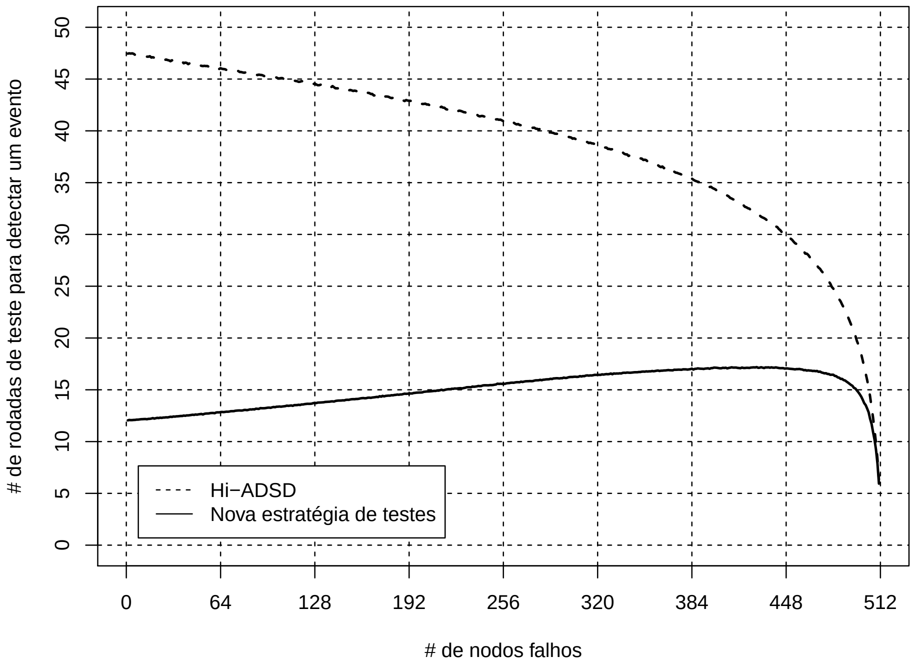

Figura 3.5: Latência média para detecção de um evento em uma rede de 512 nodos.

CAPÍTULO 4

EXPERIMENTOS NO PLANET-LAB

A necessidade de desenvolvimento de novas tecnologias e aplicações para redes, em particular a Internet, é constante. Para avaliar essas aplicações é necessário realizar experimentos em uma infraestrutura de rede que reflita o ambiente real de uso. O Planet-Lab [2] é uma plataforma global para executar e avaliar protocolos e serviços para Internet sob condições realistas. Ele já foi usado para avaliar uma grande diversidade de serviços em escala planetária, incluindo distribuição de conteúdos, DHTs (*Distributed Hash Table*), serviços de DNS (*Domain Name System*) robusto, distribuição de arquivos em larga escala, diagnóstico de falhas e anomalias, notificação de eventos, entre outros.

O PlanetMon [23] é um arcabouço para administrar e monitorar experimentos no Planet-Lab. O PlanetMon é um portal que utiliza uma arquitetura centralizada, onde um servidor central é responsável por se conectar a todos os nodos onde o experimento deve ser realizado, enviar a aplicação do usuário, enviar comandos para executar o experimento e coletar arquivos de log. Como o número de nodos que participam de um experimento pode ser grande, esta abordagem apresenta problemas de escalabilidade quando vários usuários precisam executar experimentos simultaneamente. Foi criada uma nova arquitetura para o PlanetMon onde o algoritmo *Virtual Hypercube* é utilizado para fornecer um *overlay* para execução de comandos. O servidor central conecta-se a apenas um subconjunto de nodos, os quais são responsáveis por testar e se conectar a outros nodos, executando o algoritmo proposto. Foram executados experimentos desta nova arquitetura e resultados são apresentados neste capítulo. O restante deste capítulo está organizado da seguinte maneira. A Seção 4.1 apresenta o PlanetMon e detalhes da integração do algoritmo proposto em sua arquitetura. A Seção 4.2 reporta os experimentos realizados para as métricas avaliadas: tempo para execução de comandos, latência de diagnóstico e número de testes executados.

4.1 PlanetMon: Arquitetura Baseada no *Virtual Hypercube*

O PlanetMon é um portal que permite realizar as tarefas essenciais para a execução de um experimento no Planet-Lab, incluindo a instalação da aplicação, sua execução e a coleta de dados. Além disso, é possível monitorar o comportamento dos nodos durante a execução do experimento, visualizando a topologia virtual criada através de um mapa. Os nodos utilizam uma API para realizar a comunicação com um gerador de mapas de topologias. Analisar o mapa de execução de uma aplicação de forma visual e em tempo real é mais simples do que recuperar arquivos de logs e cruzar informações. Este mapa corresponde ao grafo que representa a topologia virtual criada; nodos do sistema são vértices e arestas correspondem a uma interação entre os nodos. A API permite que o usuário associe sentido a uma interação entre nodos, sendo possível personalizar sua cor, opacidade e espessura. Por exemplo, uma aresta pode significar uma conexão entre os vértices.

O PlanetMon faz a monitoração de experimentos utilizando o Sistema de Diagnóstico Instantâneo (SDI [22]). O SDI provê um modo de executar *scripts* em uma rede ou na Internet, e cria páginas web contendo informações sumarizadas sobre os resultados dos *scripts*. A Figura 4.1 mostra um diagrama desta abordagem. O servidor central cria N conexões, uma para cada nodo do sistema. Através destas conexões são enviados comandos e suas respostas são retornadas ao servidor. Cada linha recebida é tratada de acordo com uma convenção, e o sistema identifica de qual comando a resposta foi gerada e realiza as operações necessárias para armazenar seu resultado, por exemplo salva os dados em disco e atualiza as páginas web relativas ao comando.

Além disso, o PlanetMon também provê uma API aos nodos que fazem parte de um experimento. Esta API permite que um mapa seja desenhado. Este mapa possui a posição geográfica onde os nodos estão localizados e as interações entre estes nodos, que são representadas por linhas. Os nodos podem utilizar esta API para enviar informações para o servidor central, as quais são representadas em uma mapa. O significado das linhas desenhadas entre os nodos depende da aplicação em execução e de quais informações são relevantes ao experimento.

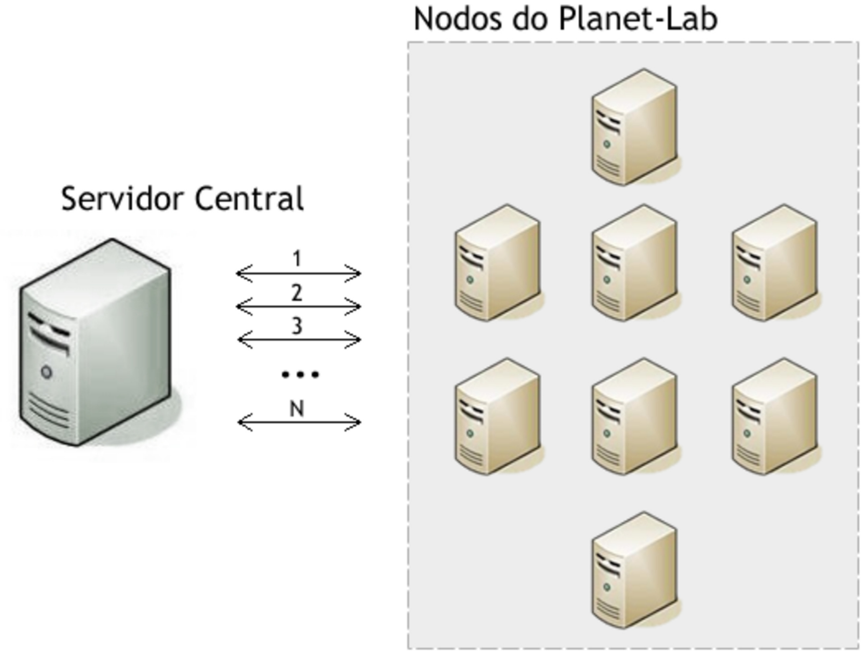

Figura 4.1: Arquitetura utilizada pelo PlanetMon.

À medida que a quantidade de nodos a serem monitorados pelo SDI aumenta, o uso de memória e processamento para manter tais conexões aumenta nas mesmas proporções. Uma grande quantidade de conexões e processos sendo executados em um servidor faz com que seu desempenho seja prejudicado. Por exemplo, experimentos utilizando o SDI em uma rede com cerca de 1800 nodos geram mais de 11000 processos no servidor, fazendo com que o escalonamento de tais processos seja prejudicado.

Como uma alternativa a esta arquitetura centralizada, o SDI foi modificado neste trabalho para criar um *overlay* utilizando os nodos a serem monitorados de forma que suas conexões sejam distribuídas de acordo com o algoritmo *Virtual Hypercube*. O servidor central não é mais responsável por criar conexões com todos os nodos da rede. Os nodos que fazem parte da rede são responsáveis por testar e se conectar uns aos outros, executando o algoritmo proposto. Em outras palavras, o algoritmo foi implementado como motor distribuído do PlanetMon. A Figura 4.2 mostra um diagrama desta arquitetura.

Cada nodo da rede recebe um identificador único. Quando um nodo é inicializado ele executa o algoritmo baseado em seu identificador, e cria conexões com os nodos que ele deve testar. Os testes são as próprias conexões. Quando uma conexão está ativa o nodo está no estado sem-falha, e quando uma conexão é interrompida o nodo correspondente é

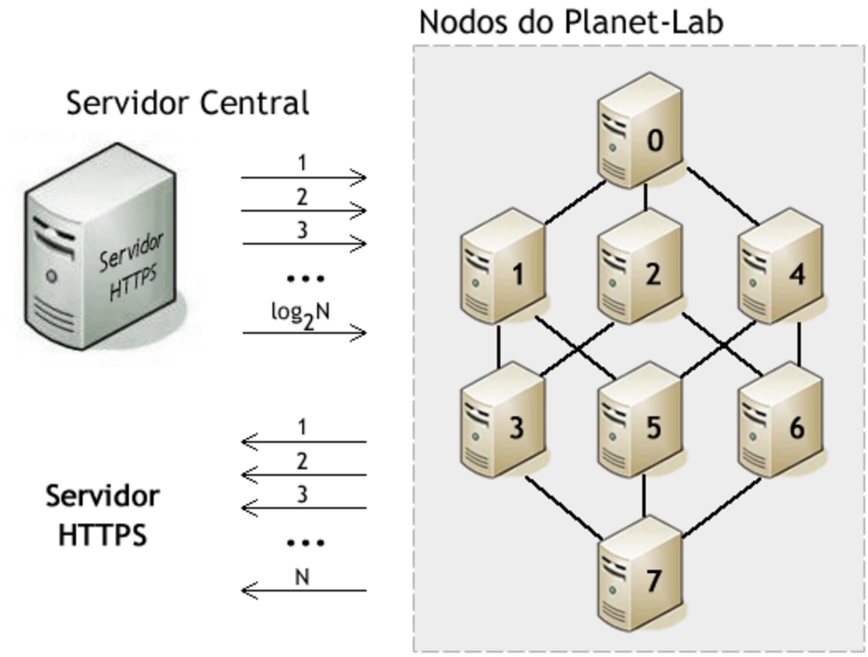

Figura 4.2: Nova arquitetura do PlanetMon utilizando o algoritmo Virtual Hypercube.

considerado falho. Conexões TCP persistentes são comumente utilizadas como estratégia de testes. As informações de diagnóstico são transmitidas através destas conexões. Vale também lembrar que o servidor central não executa o algoritmo e consequentemente não faz parte do *overlay*, ou seja, sua responsabilidade é de apenas conectar-se a um subconjunto de nodos os quais executam o algoritmo.

Para inicializar o *overlay*, o servidor central cria conexões com nodos escolhidos aleatoriamente. Ele pode se conectar a um ou mais nodos, os quais irão se conectar aos nodos que devem testar, iniciando assim a formação do *overlay*. Escolhemos sempre manter $\log_2 N$ conexões no servidor central. Quando uma conexão do servidor central com um nodo do *overlay* é perdida, um novo nodo é escolhido aleatoriamente para substituir esta conexão. O envio de comandos é realizado a partir do servidor central para os nodos diretamente conectados a ele. Estes nodos irão, então, repassar este comando para seus vizinhos, fazendo com que todos os nodos conectados recebam a mensagem. Como existem vários caminhos na rede pelos quais um nodo pode receber um comando, cada comando recebe um identificador para que seja possível checar se tal comando já foi executado e repassado para seus vizinhos. Em caso positivo, a mensagem é simplesmente ignorada.

Outra diferença significativa para a arquitetura original do SDI é a maneira como os dados são reportados para o servidor central. Na abordagem original, na qual o servidor central conecta-se com todos os nodos a serem monitorados, os resultados dos comandos executados nos nodos são enviados através da própria conexão. Na nova arquitetura estes resultados são enviados diretamente ao servidor central através do protocolo HTTPS. Esta escolha foi feita para diminuir a carga de rede nos nodos do *overlay* e melhorar o desempenho no caso geral. Caso as mensagens de resposta fossem roteadas pelo *overlay* até o servidor central, nodos que estivessem mais próximos do servidor central seriam sobrecarregados. Como todas as informações devem alcançar de alguma maneira o servidor central, enviar as informações diretamente a ele é mais eficiente do que enviar através de terceiros.

A implementação do servidor HTTPS foi realizada utilizando a plataforma Node.Js [1]. Esta plataforma foi construída sobre a *engine* JavaScript do navegador Google Chrome com o objetivo de criar aplicações de redes rápidas e escaláveis. A plataforma é baseada em eventos e tem um modelo de I/O não bloqueante que a torna leve e eficiente, sendo apropriada para aplicações de tempo real distribuídas. Além do uso da segurança do protocolo HTTPS, uma autenticação HTTP básica também é utilizada. As credenciais para esta autenticação são geradas especificamente para cada usuário. Esta autenticação evita que o servidor processe mensagens não desejadas.

Utilizar o servidor HTTPS traz várias vantagens com relação à arquitetura anterior. Uma conexão HTTP é mais barata que uma conexão SSH. Conexões são realizadas apenas quando existem dados para serem transferidos. Apenas um processo é utilizado para receber todas as requisições, liberando o sistema operacional de escalonar uma quantidade grande de processos.

O uso do servidor HTTPS para receber dados vindos dos nodos da rede fez com que uma grande parte dos *scripts* utilizados pelo SDI fossem reescritos. A geração de arquivos XML para a interface web era realizada a partir de vários arquivos em disco, os quais devem ser manipulados para que algum dado seja alterado na interface web. Com o novo servidor, os arquivos XML são gerados de uma maneira mais flexível, baseado em uma estrutura de dados interna. As informações que acabam de ser recebidas são mantidas em memória, porém também continuam sendo armazenadas em disco mas de uma forma não bloqueante. Como as informações mais recentes ficam na memória do servidor a criação dos arquivos XML, que permitem a visualização dos dados na web, é muito rápida.

Para que um nodo seja capaz de executar o algoritmo e criar conexões com outros nodos da rede ele precisa de várias informações, as quais são inicialmente fornecidas pelo servidor central. Dentre elas estão o valor de N, o identificador do nodo, a lista com identificadores e endereços de todos os nodos da rede, o endereço e credenciais do servidor HTTPS que deve receber os dados gerados, a chave privada necessária para acessar outros nodos, os comandos necessários para inicializar um novo nodo e criar uma nova conexão, e os comandos que devem ser executados assim que a conexão seja estabelecida. Após o servidor central repassar estas informações para os primeiros nodos, estes passam a ser responsáveis por compartilhar estes dados quando criarem novas conexões. Todas estas informações são armazenadas na memória de cada nodo, e todas as comunicações são realizadas através de conexões SSH.

Como um nodo pode receber conexões de vários outros nodos é preciso garantir que apenas uma instância do algoritmo é executada em um dado momento. Assim, como quando uma conexão cujos processos estavam executando o algoritmo deixa de funcionar, é preciso que outro processo assuma a responsabilidade de executar o algoritmo para aquele nodo.

Durante a execução do algoritmo, os testes são implicitamente baseados nos estados da conexão entre o nodo testador e o nodo testado. Por exemplo, se o nodo i precisa testar o nodo j, o nodo i verifica se ele tem uma conexão aberta com o nodo j. Em caso positivo, o nodo j é considerado sem-falha. Caso contrário, uma conexão é criada com o nodo j, e seu estado será atualizado de acordo com o sucesso desta conexão. Sempre que uma conexão é estabelecida ou perdida, o *timestamp* correspondente é atualizado.

Sempre que um teste é realizado, o vetor de *timestamps* do nodo testado é transferido ao nodo testador. O nodo testador então é responsável por verificar se algum dos *timestamps* recebidos são mais recentes do que os *timestamps* conhecidos por ele, atualizando aqueles que são necessários.

Para finalizar a implementação da nova arquitetura do PlanetMon foi necessária uma re-implementação do servidor da API utilizada para geração de mapas. A nova implementação utiliza o mesmo servidor HTTP que provê arquivos XML para a interface web, para prover os arquivos XML necessários para a geração dos mapas. Além disso, foi implementada uma interface de visualização utilizando WebSockets [26]. Basicamente WebSockets permitem que uma conexão TCP *full-duplex* seja estabelecida entre o browser do cliente e o servidor, permitindo uma comunicação direta entre ambos. Esta tecnologia melhora consideravelmente o desempenho de aplicações em tempo real. Outras estratégias para implementação de aplicações de tempo real na web, como *polling*, *long-polling* e *streaming*, impõem uma sobrecarga muito grande na comunicação. Utilizando WebSockets as atualizações realizadas nos mapas através da API do PlanetMon são enviadas em tempo real ao browser do usuário que observa o comportamento da sua aplicação.

4.2 Experimentos

Para validar a solução e avaliar se a nova arquitetura é adequada para a execução e monitoração de experimentos no Planet-Lab, vários experimentos foram executados. A Subseção 4.2.1 mostra os experimentos realizados para medir o desempenho do *overlay* ao entregar comandos aos nodos da rede. A Subseção 4.2.2 exibe os experimentos realizados para medir a latência e a quantidade de testes que o algoritmo *Virtual Hypercube* utiliza para detectar os eventos na rede.

Todos os experimentos foram realizados em um conjunto de 64 nodos do Planet-Lab, porém durante os experimentos um conjunto de 10 nodos se manteve falho. O tempo utilizado para o intervalo de testes foi de trinta segundos.

4.2.1 Execução de Comandos

Como o servidor central está conectado a no máximo $\log_2 N$ nodos em um certo instante de tempo, não é possível enviar um comando diretamente a todos os nodos pertencentes ao *overlay*. Como dito anteriomente, os nodos que recebem um comando são responsáveis por enviar este comando a todos os seus vizinhos, ou seja, todas as suas conexões ativas no momento. Esta estratégia faz com que um nodo receba o mesmo comando mais de uma vez, porém como cada comando tem um identificador único é possível distinguir se um comando recebido deve ser executado e repassado ou não.

Este experimento foi realizado para medir a quantidade de tempo necessária para que um comando seja propagado para toda a rede. Para isso o comando utilizado neste experimento consiste em enviar uma requisição HTTP para o servidor central contendo o identificador que o executou. O servidor então é responsável por contabilizar a quantidade de tempo desde o envio do comando até o recebimento destas requisições.

Foram testados dois cenários. O primeiro consiste em o servidor central se conectar a $\log_2 N$ nodos diretamente, o que é o comportamento da arquitetura proposta. No segundo cenário o servidor central conecta-se diretamente a apenas 1 nodo. Os gráficos das figuras 4.3 e 4.4 mostram a quantidade de nodos que executou o comando com sucesso ao longo do tempo para os dois cenários experimentados. Para cada cenário foram executados 80 experimentos, e os resultados mostrados nos gráficos são o tempo médio, máximo e mínimo destas execuções.

Como existem muitos caminhos para que o comando alcance um certo nodo, aquele caminho que apresentar maior desempenho no momento será o que entregará o comando ao destino. A diferença de desempenho entre os dois cenários mostra que assim que o comando é recebido por um dos nodos da rede a propagação para o restante da rede é muito similar. Podemos perceber também que os valores máximo e mínimo estão mais próximos da média quando o servidor está conectado a mais de um nodo. Isso acontece pois o desempenho não é prejudicado por nodos carregados, o que pode acontecer mais facilmente quando existe uma conexão com apenas um nodo, onde a propagação depende muito deste nodo e de seus vizinhos.

4.2.2 Latência e Número de Testes

Para validar a implementação do algoritmo em um ambiente real é necessário verificar a latência para a detecção de um evento e a quantidade de testes executados pela rede

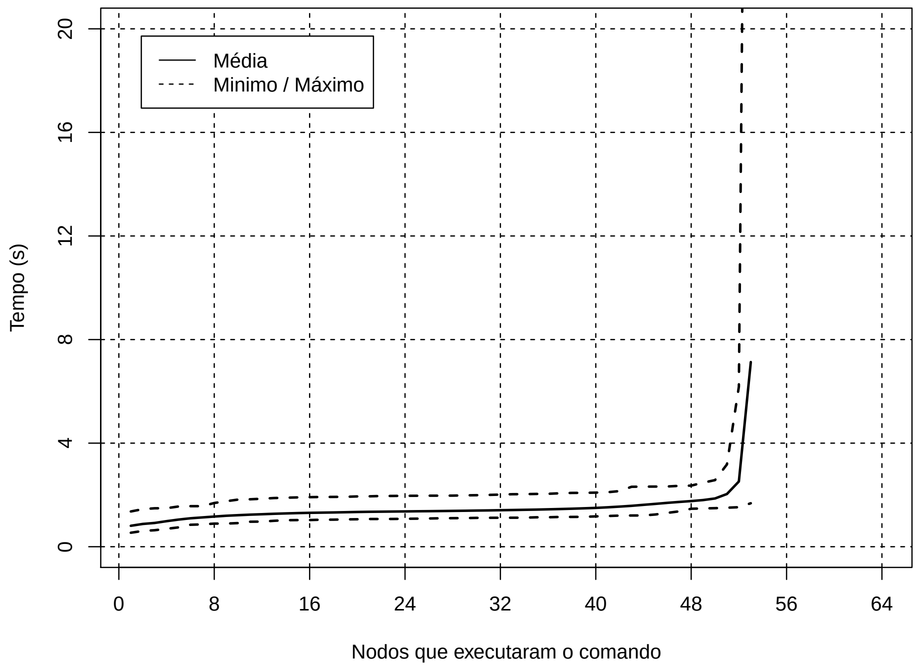

Figura 4.3: Tempo necessário para que os nodos do *overlay* executem um comando quando o servidor central mantém uma conexão.

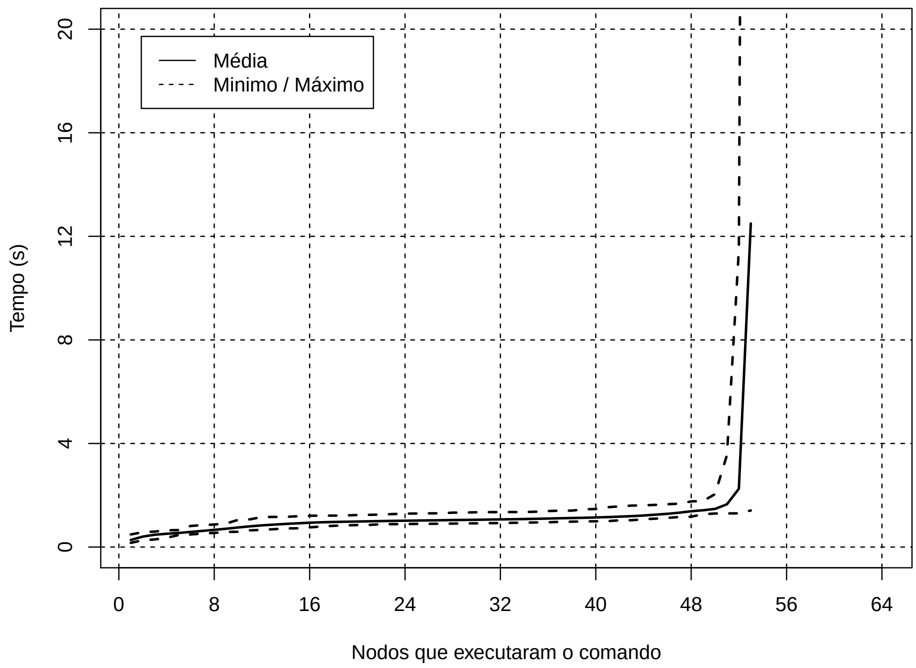

Figura 4.4: Tempo necessário para que os nodos do *overlay* executem um comando quando o servidor central mantém $\log_2 N$ conexões.

durante este período. Para isso, foram realizados experimentos com injeção de falhas seguida de recuperação de nodos da rede, e verificou-se quantas rodadas de testes e quantos testes foram necessários para que todos os nodos da rede detectassem tal evento.

Para obter estas medidas um comando foi criado para recuperar informações sobre o estado de execução do algoritmo. Este comando é responsável por reportar quantas rodadas de testes foram executadas e quantos testes foram realizados deste o início da execução do algoritmo, e também o vetor de *timestamps*. Estas informações são então enviadas a um servidor HTTP que é responsável por calcular os dados de latência e número de testes.

Foram executados um total de 60 experimentos. Para cada experimento a média da quantidade de rodadas de testes executadas e do número de testes realizados foram calculadas. Os resultados mostrados são as médias entre estas execuções. Cada execução consiste nos seguintes passos. Primeiramente o servidor tem sua memória liberada, assim não tendo informações sobre nenhum nodo da rede. O comando citado anteriormente é enviado através do *overlay*, e o servidor recebe e armazena os dados para consulta posterior. Um falha é injetada em um dos nodos, fazendo com que todos os nodos que antes estavam conectados a ele detectem que este nodo está falho. Após isso, na próxima rodada de testes de um destes nodos, uma nova conexão será criada e o nodo que antes estava falho agora será considerado sem-falha. Durante este processo o *timestamp* relacionado será incrementado de 2.

O comando para recuperar as informações sobre o estado de execução do algoritmo é enviado em um intervalo de tempo pequeno, e o servidor HTTP recebe estes dados e reporta os nodos que foram capazes de detectar esta mudança de *timestamp*. Como as informações sobre número de rodadas de testes e quantidade de testes executados são armazenadas antes da execução do evento, caso um nodo tenha detectado o evento corretamente o servidor então compara estes valores.

A Tabela 4.1 mostra os resultados dos experimentos. Na média foram necessárias aproximadamente 2.69 rodadas de teste para que os nodos detectassem um evento, e estes fizeram uma média de aproximadamente 16.73 testes durante este período. Vale lembrar que durante os experimentos não existe nenhuma garantia de sincronicidade entre os nodos, ou seja, em um instante de tempo é possível que os nodos estejam realizando testes em *clusters* de tamanhos diferentes. Por este motivo as estatísticas recolhidas com relação à quantidade de testes dependem do estado atual de execução do algoritmo na rede. Como vários nodos estavam falhos durante a execução do experimento a quantidade de testes realizados por um nodo durante suas rodadas pode ser superior a $\log_2 N$, fazendo com que a média de números de testes seja um pouco superior aos esperados $2.69 * \log_2 N$ testes. O mesmo acontece com a quantidade de testes executados na rede como um todo. A Figura 4.5 mostra um histograma das latências detectadas na rede.

Latência média (rodadas de teste)	2.696589
Número de testes médio por nodo	16.73348
Número de testes médio executados na rede	878.8254

Tabela 4.1: Resultados dos experimentos de latência e número de testes.

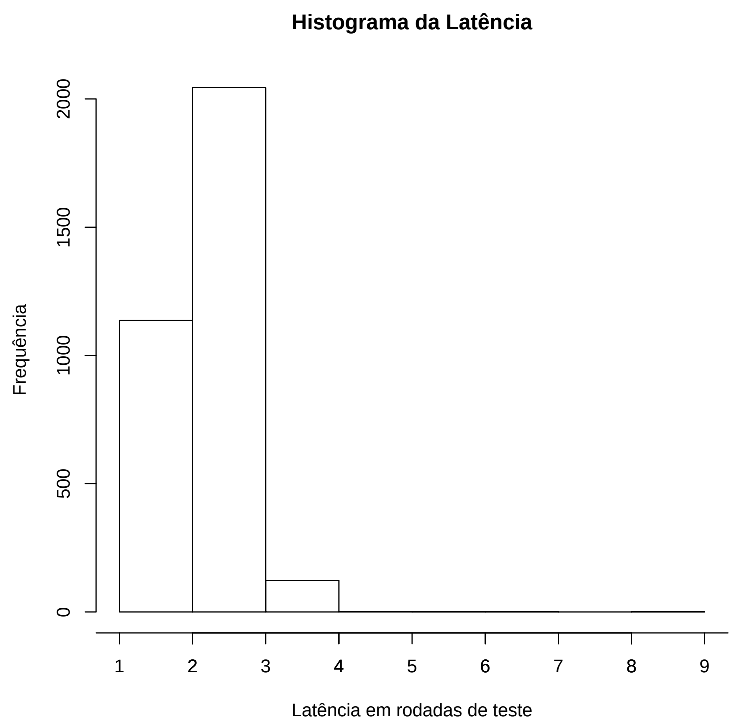

Figura 4.5: Histograma das latências detectadas durante os experimentos no Planet-Lab.

CAPÍTULO 5

CONCLUSÃO

Este trabalho apresentou o novo algoritmo *Virtual Hypercube* para o diagnóstico distribuído hierárquico que garante propriedades logarítmicas tanto para sua latência quanto para a quantidade máxima de testes executados. Em comparação com outro algoritmo de diagnóstico distribuído hierárquico, o *Hi-ADSD*, o algoritmo proposto neste trabalho reduz o número máximo de testes executados, mantendo o mesmo pior caso para a latência. Enquanto no *Hi-ADSD* o número de testes executados no pior caso é quadrático, no algoritmo proposto o limite máximo é de $N \log_2 N$ testes a cada $\log_2 N$ rodadas de testes. Ambos os algoritmos têm latência máxima de $\log_2^2 N$ rodadas.

O algoritmo *Virtual Hypercube* organiza a execução de testes em *clusters* progressivamente maiores. Na estratégia de testes proposta os testes são realizados com base na seguinte regra: um nodo i na rodada de testes s deve testar o nodo $j \in c_{i,s}$ se, e somente se, o nodo i é o primeiro nodo sem-falha em uma lista ordenada de nodos que podem testar o nodo j na rodada de teste s. Quanto um nodo sem-falha é testado informações de diagnóstico sobre todo o sistema são enviadas. Uma nova função $c_{i,s}$ foi apresentada para representar tanto os nodos que o nodo i pode testar na rodada de testes s quanto os nodos que podem testar o nodo i nesta mesma rodada.

Resultados de simulação inclusive comparando o novo algoritmo com o algoritmo *Hi-ADSD* foram apresentados. Com relação ao número de testes o novo algoritmo apresentou uma quantidade logarítmica de testes em cenários onde o algoritmo *Hi-ADSD* apresentou comportamento quadrático. A latência média do novo algoritmo foi inferior ao algoritmo *Hi-ADSD* em todos os cenários, devido tanto à nova estratégia de testes quanto à obtenção de informações de diagnóstico utilizando *timestamps*.

Uma implementação do algoritmo *Virtual Hypercube* foi integrada ao PlanetMon, um arcabouço para execução e monitoramento de experimentos no Planet-Lab. Uma nova arquitetura para o PlanetMon foi criada, onde os nodos que fazem parte de um experimento utilizam o novo algoritmo para criar um *overlay* utilizado como motor distribuído para execução de comandos. Experimentos foram realizados para medir o tempo necessário para executar comandos no *overlay*. Além disso foram medidas a quantidade de testes realizados e latência para detecção de falhas na rede. Os resultados mostraram que a execução do algoritmo em um ambiente realista é viável, visto que a latência média e a quantidade de testes realizados condizem com os limites teóricos provados pelo trabalho.

Trabalhos futuros incluem comparações com os algoritmos de diagnóstico *DiVHA*, *Hi-ADSD with Detours* e *Hi-ADSD with Timestamps*, que podem ser feitas para explorar detalhes de desempenho dos diversos algoritmos em situações particulares. Examinar o funcionamento do novo algoritmo em condições mais relaxadas é necessário para analisar seu desempenho em um ambientes onde certas asserções não são respeitadas. Além disso, é possível trabalhar em uma especificação do algoritmo que permite que cada nodo execute testes nos $\log_2 N$ clusters em uma única rodada, mantendo um custo razoável (logarítmico) e, ao mesmo tempo, baixando a latência máxima para $\log_2 N$ rodadas de teste ao invés de $\log_2^2 N$ no pior caso. Reduzir a computação necessária para a execução do algoritmo também deve ser considerada para sua execução em redes com um número grande de nodos. Diversas otimizações são possíveis para evitar que a toda rodada de testes os testadores executem múltiplas verificações para determinar exatamente quais testes devem realizar. Outra linha de trabalho pode expandir a definição do algoritmo para que seja possível realizar diagnóstico em redes de topologia arbitrária.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Node.js. `http://www.nodejs.org`. Acessado: maio de 2013.

[2] Planet-lab. `http://www.planet-lab.org`. Acessado: maio de 2013.

[3] F. Barsi, F. Grandoni, e P. Maestrini. A theory of diagnosability of digital systems. *IEEE Transactions on Computers*, 1976.

[4] R.P. Bianchini Jr. e R.W. Buskens. Implementation of online distributed system-level diagnosis theory. *IEEE Transactions on Computers*, 1992.

[5] L.C.E. Bona, K.V.O. Fonseca, E.P. Duarte Jr., e S.L.V. Mello. Hyperbone: A scalable overlay network based on a virtual hypercube. *IEEE International Symposium on Cluster Computing and the Grid*, 2008.

[6] A.T. Dahbura, K.K. Sabnani, e L.L. King. The comparison approach to multiprocessor fault diagnosis. *IEEE Transactions on Computers*, 1987.

[7] E.P. Duarte Jr., L.C.P. Albini, A. Brawerman, e A.L.P. Guedes. A hierarquical distributed fault diagnosis algorithm based on clusters with detours. *Network Operations and Management Symposium*, 2009.

[8] E.P. Duarte Jr., A. Brawerman, e L.C.P. Albini. An algorithm for distributed hierarchical diagnosis of dynamic fault and repair events. *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, 2000.

[9] E.P. Duarte Jr. e T. Nanya. A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Transactions on Computers*, 1998.

[10] E.P. Duarte Jr., A. Weber, e K.V.O. Fonseca. Distributed diagnosis of dynamic events in partitionable arbitrary topology networks. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[11] E.P. Duarte Jr., R.P. Ziwich, e L.C.P. Albini. A survey of comparison-based system-level diagnosis. *ACM Computing Surveys*, 2011.

[12] M. Elhadef e A. Nayak. Comparison-based system-level fault diagnosis: A neural network approach. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[13] S.L. Hakimi e A.T. Amin. Characterization of connection assignment of diagnosable systems. *IEEE Transactions on Computers*, 1974.

[14] S.L. Hakimi e K. Nakajima. On adaptive system diagnosis. *IEEE Transactions on Computers*, 1984.

[15] S.H. Hosseini, J.G. Kuhl, e S.M. Reddy. A diagnosis algorithm for distributed computing systems with dynamic failure and repair. *IEEE Transactions on Computers*, 1984.

[16] L.E. LaForge, K.F. Korver, e M.S. Fadali. What designers of bus and network architectures should know about hypercubes. *IEEE Transactions on Computers*, 2003.

[17] P. Lai, J.J.M. Tan, C. Chang, e L. Hsu. Conditional diagnosability measures for large multiprocessor systems. *IEEE Transactions on Computers*, 2005.

[18] M.H. MacDougall. *Simulating computer systems: techniques and tools*. MIT Press, Cambridge, MA, USA, 1987.

[19] J. Maeng e M. Malek. A comparison connection assignment for self-diagnosis of multiprocessor systems. *Symposium on Fault-Tolerant Computing*, 1981.

[20] G.M. Masson, D.M. Blough, e G.F. Sullivan. Fault-tolerant computer system design. capítulo System diagnosis, páginas 478–536. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[21] F.P. Preparata, G. Metze, e R.T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, 1967.

[22] B.C. Ribas, D.G. Pasqualin, e V.K. Ruoso. Sdi - sistema de diagnóstico instantâneo. *Workshop de Software Livre*, 2009.

[23] V.K. Ruoso, L.C.E. Bona, e E.P. Duarte Jr. Planetmon: Um arcabouço para execução e monitoração de experimentos no planet-lab. *7o Workshop de Redes Dinâmicas e Sistemas Peer-to-Peer (WP2P), 29o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, 2011.

[24] M. Serafini, A. Bondavalli, e N. Suri. Online diagnosis and recovery: On the choice and impact of tuning parameters. *IEEE Transactions on Dependable and Secure Computing*, 2007.

[25] A. Subbiah e D.M. Blough. Distributed diagnosis in dynamic fault environments. *IEEE Transactions on Parallel and Distributed Systems*, 2004.

[26] V. Wang, F. Salim, e P. Moskovits. *The Definitive Guide to HTML5 WebSocket*. Apressus Series. Apress, 2012.