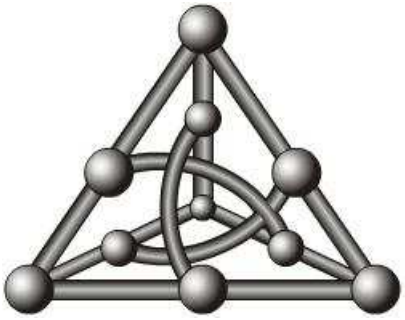
Detecção Automática de Anomalias em Ambientes Distribuídos Utilizando Redes Bayesianas

Brivaldo Alves da Silva Junior

Dissertação de Mestrado

Orientação: Prof. Ronaldo Alves Ferreira

Área de Concentração: Redes de Computadores



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
4 de março de 2013

Resumo

Diagnosticar anomalias em grandes redes corporativas consome tempo considerável das equipes de suporte técnico, principalmente pela complexidade das inúmeras interações existentes entre as aplicações e os elementos de rede (servidores, roteadores, enlaces, etc.).

Nos últimos anos, vários trabalhos científicos propuseram ferramentas automatizadas para detecção de anomalias em ambientes distribuídos. As ferramentas são divididas em dois grandes grupos: as que usam abordagens intrusivas, em que as aplicações precisam ser alteradas para registrar eventos de comunicação e facilitar o rastreamento de problemas; e as não intrusivas, em que pacotes são capturados diretamente da rede e técnicas estatísticas são aplicadas para inferir, com um certo grau de confiança, as possíveis causas dos problemas. As duas abordagens possuem vantagens e desvantagens. Entretanto, as técnicas não intrusivas são mais aceitas pela facilidade de implantação e também por não exigirem que aplicações já desenvolvidas sejam alteradas para incluir mecanismos de registro de eventos.

A abordagem mais completa e promissora para a solução desse problema, denominada Sherlock, utiliza traços de rede para construir automaticamente um Grafo de Inferência (GI) que modela as múltiplas interações e dependências presentes em um ambiente distribuído. Apesar do progresso feito por Sherlock na modelagem do problema, o seu tempo de execução para inferir as possíveis causas e a precisão dos resultados de detecção das anomalias ainda deixam a desejar.

Este trabalho propõe Nemo, uma ferramenta que explora conhecimento específico do domínio do problema e uma propriedade teórica de Redes Bayesianas para reduzir significativamente um GI e, consequentemente, o tempo de execução. Resultados de simulação utilizando dados reais e sintéticos mostram que Nemo reduz o tempo de execução de Sherlock em mais de 90% e melhora sua precisão em todos os cenários simulados. Além disso, este trabalho também apresenta uma extensa revisão bibliográfica sobre o assunto e comparações qualitativas dos principais métodos propostos na literatura.

Palavras-Chave:*gerenciamento de redes, sistemas distribuídos, inferência estatística, detecção de anomalias, mapeamento de rede.*

Abstract

Diagnosing anomalies in large enterprise networks consumes significant time of technical support teams, mainly because of the numerous complex interactions among the applications and network elements (servers, routers, links, etc.).

In recent years, several studies have proposed scientific tools for automated detection of anomalies in distributed environments. The tools are divided into two major groups: those that use intrusive approaches in applications which need to be changed to record communication events and facilitate tracking of problems, and those that are non intrusive, in which packets are captured directly from the network and statistical techniques are applied to infer, with a degree of confidence, the possible causes of problems. Both approaches have advantages and disadvantages. However, non-intrusive techniques are more accepted by their ease of deployment and also because they do not require that applications mechanisms be modified to include registration events.

The most complete and promising approach for solving this problem, called Sherlock, uses network traces to automatically build an Inference Graph (IG) that models the multiple interactions and dependencies that are present in a distributed environment. Despite the progress provided by Sherlock in the problem modeling, its execution time for inferring the probable causes and the precision of its anomaly detection results leave much room for improvements.

This work proposes Nemo, a tool that explores domain-specific knowledge and a theoretical property of Bayesian Networks to significantly reduce the IG and consequently the execution time. Simulation results using real and synthetic data show that Nemo reduces Sherlock's execution time by over 90% and improves its precision in all simulated scenarios. Moreover, this work also presents an extensive survey on the subject and qualitative comparisons of the main methods proposed in the literature.

Keywords: *network management, distributed system, statistical inference, anomaly detection, network mapping.*

Conteúdo

1 Introdução **1**

1.1 Organização do Texto 3

2 Conceitos Básicos e Ferramentas Tradicionais **4**

2.1 Modelos de Programação e de Serviços de Rede 5

2.1.1 Processos e *Threads* 5

2.1.2 Select 6

2.1.3 SEDA 7

2.1.4 Cliente-Servidor 7

2.1.5 Peer-to-Peer 8

2.2 Protocolo de Gerenciamento SNMP 8

2.3 Ferramentas Tradicionais de Análise de Rede 9

2.3.1 traceroute 9

2.3.2 ping 10

2.3.3 nmap 10

2.3.4 Nagios 10

2.4 Considerações Finais 11

3 Métodos Intrusivos de Detecção de Anomalias **12**

3.1 Magpie 12

3.2 Pinpoint 15

3.3 PIP 18

3.4 X-Trace 20

3.5 vPath 24

3.6 Comparação Qualitativa dos Métodos Intrusivos 26
3.7 Outros Métodos Intrusivos 27
3.7.1 Whodunit 27
3.7.2 BorderPatrol 28
3.7.3 Macroscope 28
3.7.4 Dapper 29
3.8 Considerações Finais 29

4 Métodos Não Intrusivos 30

4.1 Project5 31
4.2 Wap5 34
4.3 Sherlock 39
4.4 eXpose 44
4.5 Constellation 47
4.6 Netmedic 49
4.7 Spotlight 53
4.8 Comparação Qualitativa dos Métodos Não Intrusivos 56
4.9 Outros Métodos Não Intrusivos 58
4.9.1 Orion 58
4.9.2 DYSWIS 58
4.9.3 SNAP 59
4.9.4 NSDMiner 59
4.10 Considerações Finais 60

5 Ferramenta de Detecção de Anomalias Nemo 61

5.1 Grafo de Dependências de Serviços 62
5.2 Grafo de Inferência de Rede 64
5.3 Redes Bayesianas 65
5.3.1 *d*-separação e *d*-conexão 66
5.4 Redução do Grafo de Inferência 68
5.5 Função de Pontuação 69

5.6 Aspectos de Implementação 71
5.7 Considerações Finais 75

6 Resultados Experimentais **76**
6.1 Metodologia de Avaliação 76
6.2 Redução do Grafo de Inferência 78
6.3 Precisão 79
6.4 Tempo de execução 81
6.5 Considerações Finais 83

7 Conclusão **84**
7.1 Trabalhos Futuros 85

A Implementação da *d*-separação em Python **90**

Lista de Figuras

2.1 Modelos de processamentos de requisições. 6

3.1 Fluxo de execução do Magpie. 13
3.2 Arquitetura do Pinpoint. 16
3.3 Propagação de IDs no Pinpoint [7]. 17
3.4 Ferramenta de depuração do PIP. 20
3.5 Formato dos campos dos metadados do X-Trace [13]. 21
3.6 Propagação de metadados do X-Trace. 22
3.7 Acesso a relatórios do X-Trace. 23
3.8 Caminho anotado no X-Trace. 24
3.9 Processamento de uma requisição do vPath 25

4.1 Caminho de uma requisição no Project5. 32
4.2 Ordem de interações em requisições do Project5. 33
4.3 Sequência de acontecimentos em uma requisição no Project5. 33
4.5 Pacotes chegando e saindo do *host X* em tempos distintos [34]. 37
4.7 Pesos normalizados como a probabilidade de um caminho ser o correto [34]. 38
4.8 Caminho causal de uma requisição de cliente. 39
4.9 Arquitetura de funcionamento de Sherlock [2]. 40
4.10 Trecho do GI gerado pelo Sherlock. 41
4.11 Tabela verdade do modelo de meta nós Ruído-Máximo. 41
4.12 Tabela verdade do modelo de meta nós Seletor. 42
4.13 Tabela verdade do modelo de meta nós *Failover* [2]. 43
4.14 Esquema geral do fluxo de trabalho do eXpose [22]. 45

4.15 Constelação criada pelo Constellation com uma hora de requisições. 48
4.16 Constelação criada pelo Constellation com duas horas de requisições. . . . 48
4.17 Estágios de processamento do serviço local do Constellation. 49
4.18 Ilustração de um problema de visibilidade do Netmedic. 51
4.19 Fluxo de funcionamento geral do Netmedic [24]. 52
4.20 Arquitetura de funcionamento de Spotlight [20]. 53
4.21 Trecho do GI gerado pelo Spotlight. 55
4.22 Compressão de um grafo de dependências para um bipartido. 55

5.1 Visão geral da ferramenta Nemo. 61
5.2 Mapeamento de dependências entre serviços. 63
5.3 Grafo gerado automaticamente por Nemo. 65
5.4 Nó v_1 como (a) colisor e (b) não colisor. 66
5.5 Regras da d-separação. 67
5.6 Exemplo de conjuntos d-separados. 67
5.7 Distribuição de tempos de resposta de um serviço monitorado por Nemo. . 71
5.8 Estrutura lógica do GI dentro do gerente. 73
5.9 Implementação inicial de um visualizador do GDS gerado por Nemo. . . . 74

6.1 Exemplo de GI com injeção de falhas. 78
6.2 Taxa de redução média de nós raízes de problema. 79
6.3 Comparação da precisão do Sherlock, Spotlight e Nemo. 80

Lista de Tabelas

3.1 Tabela qualitativa dos métodos intrusivos. 27

4.1 Tabela qualitativa dos métodos não intrusivos. 57

5.1 Tempo médio para cálculo de números primos. 73

6.1 Taxa de detecção correta de pelo menos uma anomalia. 80
6.2 Percentual de acertos do Sherlock com modificações do Nemo. 81
6.3 Tempo médio de execução das três abordagens. 82
6.4 Tempo médio do Sherlock com intervalo de confiança de 95%. 82
6.5 Tempo médio do Spotlight com intervalo de confiança de 95%. 83
6.6 Tempo médio do Nemo com intervalo de confiança de 95%. 83

Lista de Algoritmos

1 d-separação (G, X, Z) 68
2 Redução (G) 69
3 Ferret (O, G, X) 70
4 Pontuação (n) 71

Lista de Siglas

AD *Active Directory*

AD *Always Down*

API *Application Programming Interface*

ASN.1 *Abstract Syntax Notation One*

AT *Always Troubled*

CPU *Central Processing Unit*

CT-NOR *Continuous Time Noisy-Or*

DA *Domínio Administrativo*

DNS *Domain Name System*

DYSWIS *Do You See What I See*

EJB *Enterprise Java Bean*

EM *Expectative Maximization*

ETW *Event Tracing for Windows*

FFT *Fast Fourier Transform*

GDA *Grafo Direcionado Acíclico*

GDS *Grafo de Dependência de Serviço*

GI *Grafo de Inferência*

HTTP *Hypertext Transfer Protocol*

ICMP *Internet Control Message Protocol*

IDT *Interruptor Descriptor Table*

IP *Internet Protocol*

IPX *Internetwork Packet Exchange*

JVM *Java Virtual Machine*

LAN *Local Area Network*

LDAP *Lightweight Directory Access Protocol*

MIB *Management Information Bases*

MSI *Microsoft System Installer*

Nemo *Network Monitor*

NTP *Network Time Protocol*

OASIS *Overlay Anycast Service Infrastructure*

OID *Object Identifiers*

P2P *Peer-to-Peer*

PCFG *Probabilistic Context Free Grammar*

RADIUS *Remote Authentication Dial In User Service*

RAID *Redundant Array of Independent Disks*

RB *Rede Bayesiana*

RMI *Remote Method Invocation*

RPC *Remote Procedure Call*

SEDA *Staged Event-Driven Architecture*

SLDG *Service Level Dependency Graph*

SNMP *Simple Network Management Protocol*

SOAP *Simple Object Access Protocol*

TCP *Transmission Control Protocol*

TI *Tecnologia da Informação*

UDP *User Datagram Protocol*

UFMS *Universidade Federal de Mato Grosso do Sul*

UML *Unified Modeling Language*

VM *Virtual Machine*

VMM *Virtual Machine Manager*

WINS *Windows Internet Name Service*

WIPSo *Web Interactions Per Second for Ordering*

Agradecimentos

Quero agradecer à minha família querida pelo apoio em tudo que decidi fazer na vida. À minha esposa, Elizângela, pelo amor, carinho, paciência e apoio em todas as etapas deste trabalho. Aos meus amigos Ismayle, Valéria, André, Carol, Diego, Evelyn, Fabrício Carvalho e todos aqueles que não citei o nome, mas que, com certeza, mantinham pensamentos positivos e estavam comigo quando precisei. Agradeço ainda aos meus colegas de trabalho do NTI Péricles, Juliano, Marcelo Miranda, Cleonice, André Lanoa, Diego Bairros e a todos que me apoiaram e não me permitiram desistir.

Todo aprendizado é doloroso, difícil, cansativo, por mais que gostemos de uma área, aprender dói. Devemos deixar o passado para trás e aprender com o presente, pois o passado cumpriu seu objetivo. Por isso, gostaria de agradecer muito ao meu orientador de mestrado, o professor Ronaldo Alves Ferreira, por se dispor a me guiar neste processo, pela paciência, pelos finais de semana que ele dispendeu para me ensinar. Como um mestre *jedi* que ensina seu *padwan* os primeiros passos na força. Não foi fácil, mas tudo que aprendi foi realmente muito útil.

Com certeza fazer mestrado trabalhando é uma experiência extremamente desafiadora. Além do tempo reduzido para estudar e dedicar, o trabalho não permite tréguas em seus problemas constantes. Mas nenhum problema é capaz de superar a vontade aliada ao apoio de familiares, amigos e colegas. Por isso, estou muito feliz por ter conseguido chegar nesse momento tão importante. Independente de qualquer coisa, aprendi muito.

Também gostaria de agradecer a professora Hana Karina Rubinsztejn pelas correções sugeridas na qualificação, ao professor Irineu Sotoma por acompanhar este trabalho na qualificação e participar da banca de defesa e ao professor Fábio Moreira Costa por participar e colaborar com correções e ajustes no trabalho apresentado nesta defesa de mestrado. E finalmente, gostaria de agradecer a FACOM (Faculdade de Computação da UFMS) pela oportunidade de fazer um mestrado de qualidade e pela dedicação dos professores em criar um ambiente saudável de aprendizado aos seus alunos.

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

Capítulo 1

Introdução

As grandes corporações dependem cada vez mais da Internet para desenvolver suas atividades cotidianas. Elas utilizam a grande rede para vender produtos, para troca de mensagens entre funcionários e parceiros, ou para divulgar suas marcas e serviços. Num passado não muito distante, uma empresa podia desenvolver suas atividades utilizando um único servidor para armazenar seus dados, executar suas aplicações e se comunicar com seus parceiros. A proliferação de atividades desenvolvidas na Internet torna necessária a construção de ambientes distribuídos mais sofisticados para dar vazão a essas novas demandas, além de levar a aplicações distribuídas mais complexas e dependentes de vários componentes da infraestrutura de comunicação. Uma aplicação Web, por exemplo, pode ser executada em um cluster com servidores interligados por inúmeros roteadores em um *Data Center*, interagir com um servidor de banco de dados e fazer consultas a vários serviços distribuídos, como *Domain Name System* (DNS), *Active Directory* (AD), *Windows Internet Name Service* (WINS), *Radius*, etc. Consequentemente, anomalias, provocadas por falhas ou sobrecargas, em um dos componentes desse novo ambiente podem levar ao mau funcionamento de uma aplicação e uma percepção de baixa qualidade pelos clientes.

Nesse novo cenário, detectar anomalias em ambientes distribuídos tem se tornado uma tarefa extremamente desafiadora. Ferramentas tradicionais de gerenciamento, como Openview [16], Tivoli [18], Nagios [14], nmap [28], ping e traceroute [19], deixam muito a desejar, pois fornecem informações de alta granularidade e dependem demasiadamente das habilidades dos administradores de rede para detecção e diagnóstico de problemas. Essas ferramentas, normalmente, fornecem apenas informações estatísticas, de conectividade ou de disponibilidade de um componente de software ou de hardware da rede, mas são incapazes de determinar os relacionamentos e as dependências entre eles, dificultando, assim, o rápido diagnóstico de um problema com base em sintomas observados pelos usuários da rede.

Nos últimos anos, surgiram várias propostas para se automatizar o processo de detecção e diagnóstico de anomalias em ambientes distribuídos [1, 2, 4, 7, 39]. As principais propostas da literatura podem ser divididas em dois grandes grupos: intrusivas e não intrusivas. Nas propostas intrusivas [4, 7, 33, 3, 39], as aplicações são modificadas e in-

strumentadas para registrar os caminhos causais, ou seja, os caminhos percorridos por uma determinada requisição. Com os caminhos causais, é possível determinar os componentes de software e hardware envolvidos no processamento de uma requisição. Com isso, uma anomalia em uma aplicação pode ser mais facilmente diagnosticada analisando-se o conjunto de componentes utilizados no processamento da requisição e não todos os componentes da rede. Algumas propostas, como o X-Trace [13], não se limitam a apenas instrumentar as aplicações, mas instrumentam toda a pilha de protocolos, aumentando, assim, a visibilidade de execução das aplicações.

Embora as abordagens intrusivas produzam resultados bem confiáveis, elas dependem fortemente de programadores para instrumentar as aplicações, além de não serem úteis para monitorar aplicações legadas. Algumas abordagens sugerem a instrumentação de *frameworks* para essas situações [4], mas nesses casos a efetividade do diagnóstico cai bastante, além de não cobrir as aplicações legadas que não utilizam *frameworks*.

As propostas não intrusivas [7, 2, 3, 39, 20] não precisam modificar as aplicações ou *frameworks* pois utilizam as informações dos traços de comunicação para inferir as causas dos problemas. As principais diferenças de cada abordagem são determinadas basicamente pelas informações analisadas, metodologias estatísticas de inferência dos relacionamentos e soluções de problemas inerentes à coleta de informações (como problemas de comunicações esporádicas). Embora a abordagem não intrusiva seja menos precisa que a intrusiva, ela é considerada mais aceitável na prática, pois não exige conhecimento prévio das aplicações e nem que elas sejam modificadas para incluir mecanismos de registro de eventos. Uma vantagem dessa abordagem é sua aplicação em ambientes legados e pouco documentados.

Este trabalho propõe Nemo (*Network Monitor*), uma ferramenta para detecção automatizada de anomalias. Esta ferramenta servirá de apoio a administradores de rede para detecção e direcionamento do processo de análise e solução de problemas. Por ser uma ferramenta automatizada, ela tem pouca ou nenhuma interação humana no processo de detecção. Além disso, ela não modifica aplicações ou protocolos e, por isso, é uma ferramenta não intrusiva. Entretanto, o processo de análise e resolução de problemas ainda depende de ações humanas.

Um dos maiores problemas no processo de automatização de detecção e diagnóstico de anomalias é a modelagem, sem conhecimento prévio, da topologia da rede física, das interações lógicas de serviços e servidores e das requisições realizadas por clientes. Algumas propostas resolvem parcialmente este problema, mas nenhuma delas está livre dos problemas existentes no processo de inferência. Como as abordagens não intrusivas utilizam coletas de pacotes e traços de rede na montagem desse quebra-cabeça, todas elas enfrentam o problema de escassez de informações para tratar dependências e relacionamentos esporádicos. Além disso, quanto maior a rede, maior a quantidade de relacionamentos e problemas. Por isso, é importante que a ferramenta seja rápida e eficiente na construção de um mapeamento lógico da rede, na detecção de anomalias e na solução de problemas de implementação como a escassez citada.

A ferramenta desenvolvida neste trabalho é baseada na melhor solução não intrusiva conhecida na literatura (Sherlock [2]). Essa solução consegue mapear eficientemente a rede e suas requisições em uma Rede Bayesiana. Redes Bayesianas são muito utilizadas

na representação de aspectos do mundo real, relacionando probabilisticamente o conceito de causa e efeito. Dessa forma, com as Redes Bayesianas, é possível modelar o ambiente de rede e suas interações. O problema, além das imperfeições do mapeamento de relacionamentos, reside também no tamanho da rede necessária para mapear todas essas informações e no tempo necessário para encontrar uma anomalia. Sherlock, por exemplo, usa um algoritmo com tempo exponencial para detectar uma anomalia nesta modelagem.

As principais contribuições deste trabalho são: um algoritmo para redução do tamanho da Rede Bayesiana utilizada no processo de detecção de anomalias, uma função de pontuação baseada em conhecimento específico do problema, que resulta em melhoria na precisão dos resultados, e uma ampla revisão bibliográfica com comparações qualitativas dos principais métodos intrusivos e não intrusivos de detecção de anomalias.

O processo de validação dos resultados obtidos pelas melhorias propostas utilizou simulação, pois, com ela, foi possível testar diversas topologias de rede. Alguns parâmetros das simulações foram obtidos a partir de dados reais coletados por agentes instalados por este trabalho na rede da UFMS. A eficiência e precisão dos resultados foram comparadas com os resultados da melhor solução não intrusiva conhecida (Sherlock [2]) e uma outra que modela o problema de forma similar (Spotlight [20]). Um protótipo capaz de ser implantado em uma rede corporativa está disponível no *link* informado na conclusão deste trabalho.

1.1 Organização do Texto

O restante deste trabalho está organizado como segue. O Capítulo 2 descreve alguns modelos de programação de serviços de rede, o protocolo de gerenciamento SNMP e algumas ferramentas tradicionais de gerenciamento. O Capítulo 3 discute os principais métodos intrusivos para detecção e diagnóstico de anomalias e o Capítulo 4 os métodos não intrusivos. O Capítulo 5 apresenta a ferramenta desenvolvida e as técnicas utilizadas em sua construção. O Capítulo 6 apresenta os resultados experimentais e o Capítulo 7 a conclusão e trabalhos futuros.

Capítulo 2

Conceitos Básicos e Ferramentas Tradicionais

Quando dois computadores estão se comunicando por meio de um protocolo, eles podem fazer isso usando diferentes técnicas de programação. Cada uma dessas técnicas tem características que podem aumentar a velocidade de comunicação, a quantidade de informações trocadas ou a quantidade de computadores que podem participar da conversa. Expandindo esse raciocínio para um *data center*, são utilizados diversos protocolos, meios físicos e computadores.

Contudo, nem sempre as comunicações ocorrem como esperado. Problemas com falhas em dispositivos de rede, corrupção de informações ou sobrecarga, podem causar transtornos no processo de comunicação. Por isso, é importante que o administrador de rede seja capaz de detectar uma falha e corrigí-la. Algumas ferramentas são normalmente utilizadas nesse processo de detecção de problemas (`ping, traceroute, nmap`). Porém, elas são limitadas e fornecem poucas informações quando o problema é um pouco mais complexo.

Quando um computador está fornecendo um serviço, um cliente, ao requisitá-lo, executa diversas operações na rede. Uma delas é disparar um acesso a um serviço. O servidor ao receber essa requisição, vai processá-la. Dependendo da forma como o serviço foi programado, várias outras requisições para outros computadores na rede podem acontecer e o processo de requisição cliente e servidor ocorre, recursivamente, até que o cliente inicial receba o retorno de sua requisição. As ferramentas citadas não conseguem acompanhar todo esse processo. Como o processo pode ser diferente de serviço para serviço, para compreender como uma requisição está sendo atendida é necessário saber as técnicas de programação empregadas no seu desenvolvimento.

Além das ferramentas, existe um protocolo criado especialmente para monitorar equipamentos de rede, o SNMP (*Simple Network Management Protocol*). Esse protocolo, descrito em detalhes na Seção 2.2, foi criado para auxiliar no monitoramento das atividades que ocorrem na rede. Esse protocolo é capaz de auxiliar o administrador de redes com informações, detecções de sobrecargas e perda de comunicação de equipamentos. Embora o SNMP possa detectar quais serviços de rede estão indisponíveis, ele é incapaz de detectar

anomalias que envolvem requisições de clientes trafegando por diversos servidores.

Neste capítulo, serão abordadas técnicas de programação de servidores, para que seja possível compreender por que é complexo mapear requisições em rede. Em seguida, será descrito o protocolo de gerenciamento SNMP, por ser importante no monitoramento da rede. Por último, serão apresentadas algumas ferramentas úteis que são utilizadas no mapeamento de problemas de rede nos dias de hoje.

2.1 Modelos de Programação e de Serviços de Rede

Para entender a dificuldade de se detectar automaticamente anomalias em ambientes distribuídos, é necessário também entender como as aplicações de rede são normalmente implementadas. Isso acontece pois cada serviço pode responder às requisições de forma diferente ou então realizar outras requisições que são necessárias para atender a primeira. Por isso, uma requisição pode gerar diversas outras antes de ser atendida.

Existem várias maneiras de se implementar uma aplicação de rede e também diversos cuidados que precisam ser tomados durante uma comunicação entre aplicações distribuídas. Questões como cálculo de temporizadores para indicação de falha e maneiras de se lidar com múltiplas conexões concorrentes devem ser cuidadosamente estudadas, pois um projeto inadequado pode levar a um baixo desempenho da aplicação, diminuindo, assim, a quantidade de requisições que podem ser processadas em um intervalo de tempo. Por outro lado, as inúmeras variações no tratamento de conexões concorrentes tornam a detecção de dependências entre aplicações mais difícil [34]. Como exemplo, podemos citar situações em que uma requisição para uma aplicação pode resultar na criação de várias *threads* e conexões com outros servidores. Nesse caso, as conexões criadas são dependentes da conexão inicial em que a requisição foi recebida.

Stevens *et al.* [40] é uma boa fonte para programação de aplicações que precisam se comunicar via rede. Em particular, o livro descreve diferentes modelos de programação para lidar com concorrência em aplicações de rede. A maneira mais prática de lidar com conexões concorrentes é utilizar artifícios do próprio sistema operacional para gerenciar as várias requisições. No projeto de um serviço de rede, os modelos de programação mais utilizados fazem uso de múltiplos processos, *threads* ou da função *select*. As subseções seguintes exploram cada um desses modelos.

2.1.1 Processos e *Threads*

Processos e *threads* são diferenciados pela quantidade de memória que utilizam e pelas informações que são copiadas e compartilhadas no momento de sua criação. Todavia, enquanto as *threads* têm sua própria pilha e compartilham os descritores do programa pai com as *threads* irmãs, os processos filhos são uma cópia igual do processo pai e, por isso, são considerados mais pesados.

Tanto processos quanto *threads* podem ser utilizados para atender requisições de forma

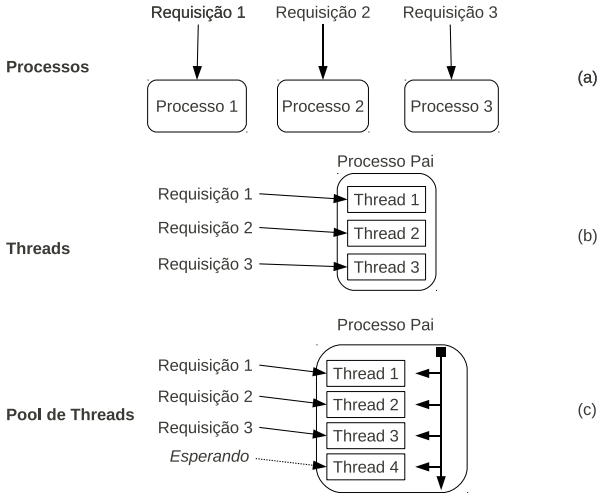


Figura 2.1: *Processamento de requisições por (a) processos, (b) threads e (c) pool de* threads.

interativa ou paralela. A ideia básica é que, para cada nova conexão um novo processo/*thread* seja criado para atendê-la. Entretanto, essa maneira simplista pode causar atrasos significativos para as aplicações, pois o atendimento da requisição será sempre precedido de um trabalho adicional do sistema operacional para criação do processo/*thread*. Para diminuir a latência de criação de threads na chegada de uma requisição, algumas aplicações criam no início de sua execução um conjunto (*pool*) de threads. Conforme as requisições chegam, as threads do *pool* são escalonadas para atendê-las. A Figura 2.1 ilustra o processamento de conexões utilizando processos, *threads* e *pool* de *threads*.

Como visto anteriormente, uma requisição pode criar outras requisições subsequentes. Por isso, o serviço pode criar uma *thread*/processo ao receber uma requisição e criar uma outra *thread*/processo para requisitar uma informação necessária à primeira. É esse o caminho que uma requisição faz dentro dos servidores que fornecem serviços. O problema está em como mapear corretamente o caminho de uma requisição quando ocorre uma falha.

2.1.2 Select

As aplicações que utilizam processos ou *threads* têm por padrão o modelo de programação com entrada e saída bloqueante. Ou seja, quando uma conexão é realizada, ela só é finalizada ao receber a resposta esperada ou um erro, ou quando o tempo de espera ultrapassa um certo limite. Isso significa que, se uma *thread* ou processo aceitar a conexão de um cliente, ela só poderá atender outro cliente quando a primeira requisição terminar. Nesse caso, a limitação do programa passa a ser a quantidade de processos ou *threads* que o sistema operacional consegue utilizar simultaneamente para atender as requisições em paralelo.

Com a utilização da função *select*, que não é bloqueante, todo o controle da conexão é

feito na própria aplicação. Isso resolve virtualmente o problema da limitação na criação de processos e *threads*. Programar serviços de rede utilizando *select* é mais complexo que o modelo de processos e *threads*, mas permite, da mesma forma, que várias requisições sejam atendidas. Isso é feito por meio de trocas de contexto dentro do serviço. Para controlar as requisições realizadas, o *select* utiliza um conjunto de filas para escalonar as prioridades. Acompanhar uma requisição em um serviço, em que o processo de paralelismo depende de como um programador desenvolveu este serviço, aumenta a complexidade no mapeamento de uma requisição.

2.1.3 SEDA

Um outro modelo de programação mais recente, e bastante interessante pela organização que é imposta à aplicação é o utilizado por *Staged Event-Driven Architecture* (SEDA) [41]. Nele, as aplicações são decompostas em estágios e cada estágio é associado a um evento que está interconectado por uma fila. A ideia é reduzir a sobrecarga associada aos modelos de concorrência baseados em *threads*, principalmente a sobrecarga associada ao custo de escalonamento causado no sistema operacional.

O modelo de SEDA utiliza controle de aceitação de requisições em cada fila e pode controlar dinamicamente os parâmetros de agendamento e carga em cada estágio. Ao quebrar os serviços em conjuntos de estágios é possível utilizar melhores técnicas de aproveitamento de código e modularidade, assim como tornar mais fácil a depuração (por módulo) de ferramentas mais complexas. Mas, infelizmente, isto torna ainda mais complexo depurar uma requisição.

2.1.4 Cliente-Servidor

Dentre todos os serviços fornecidos em uma rede de computadores, o modelo cliente-servidor é um dos mais utilizados. Isso acontece por ser o modelo mais simples de se programar, embora não seja, necessariamente, o mais eficiente em todos os casos. Nesse modelo, o serviço fica em estado de espera enquanto vários clientes realizam requisições. Dependendo de como o serviço foi programado, a conexão pode ser bloqueante e atender ou não vários clientes simultâneos.

Independente de como o serviço vai se comportar, os clientes desejam sempre que suas requisições sejam atendidas o mais rápido possível. Se o modelo de programação adotado utilizar o conceito de *threads* ou processos (o que é o mais comum), existirá uma limitação relacionada ao servidor e ao sistema operacional em execução. Por isso, um único servidor pode ser incapaz de atender uma demanda muito grande de requisições. Para sanar este problema, vários artifícios podem ser utilizados, como, por exemplo, o balanceamento de carga. Utilizando balanceamento de carga, um serviço pode ser configurado em vários servidores e estes, podem dividir entre si o total de requisições realizadas. Desta forma, o modelo cliente-servidor pode ser utilizado para manter serviços que têm alta demanda de requisições. Além disso, o fato do serviço estar espalhado em vários servidores garante maior disponibilidade.

Agora é possível entender porque mapear uma requisição é difícil. Ela pode percorrer diversos servidores ou ser tratada pelo mecanismo de balanceamento de carga antes de ser atendida. Além disso, é preciso saber quais os servidores envolvidos em uma mesma requisição. O problema inicial agora possui concorrência local e entre servidores.

2.1.5 Peer-to-Peer

Outro modelo de rede que ganhou força ultimamente foi o *Peer-to-Peer* (P2P). Esse modelo é parecido com o cliente-servidor, pois no P2P um cliente pode também ser um servidor. A diferença do P2P com o modelo cliente-servidor tradicional é que no P2P não é necessário um servidor central orquestrando as comunicações entre os clientes. Várias técnicas foram criadas para tornar o P2P mais eficiente e, por outro lado, mais complexo. Como a detecção de uma anomalia em serviços cliente-servidor é difícil, esta complexidade aumenta mais ainda quando não existe um servidor central capturando todas as interações com um serviço. É preciso compreender como os clientes atuam como servidores e como funciona a descentralização do P2P para conseguir mapear possíveis problemas em requisições.

2.2 Protocolo de Gerenciamento SNMP

O protocolo de gerenciamento de redes *Simple Network Management Protocol* (SNMP) foi concebido para coletar informações e modificar o estado de variáveis de dispositivos de rede. Ele ainda é uma das formas mais utilizadas de gerenciamento, pois os dispositivos de rede, na sua grande maioria, o implementam em pelo menos uma de suas versões (1, 2 e 3). As informações são disponibilizadas e armazenadas em *Management Information Bases*) (MIBs), descritas na linguagem *Abstract Syntax Notation One* (ASN.1).

O SNMP possui três tipos de comandos operacionais: GET, SET e GET-NEXT; os dois primeiros operam como descrito pelo próprio nome (recebendo e enviando informações) e o último é utilizado para iterar sobre as variáveis retornadas em uma consulta anterior.

A utilização do SNMP para gerenciamento de uma rede se dá por meio de um servidor administrativo (gerente) que tem a tarefa de monitorar e controlar um grupo de dispositivos de rede. O protocolo SNMP deve ser implementado em cada dispositivo monitorado e as informações devem ser disponibilizadas ou recebidas por meio de agentes. A coleta de informações desses dispositivos é realizada pelo SNMP por meio dos comandos GET ou GET-NEXT. Quando existe a necessidade de configuração remota, o comando SET é utilizado.

Outra funcionalidade importante do SNMP são os gatilhos (*traps*). Os gatilhos são configurados nos equipamentos para reagir ativamente em determinadas situações. Por exemplo, a utilização massiva de uma porta de um *switch* pode gerar um alerta. Esse alerta é enviado para o gerente pelo agente que executa no *switch*.

O protocolo SNMP não define quais informações são gerenciadas. As informações estão organizadas em MIBs (padronizadas em grupos). As MIBs descrevem a estrutura organizacional dos dados que podem ser manipulados. Em sua organização, é utilizada uma hierarquia de nomes contendo Identificadores de Objetos (*Object Identifiers* - OID), em que cada OID identifica uma variável que pode ser lida ou definida via SNMP.

A primeira versão do SNMP, conhecida como SNMPv1, é a implementação inicial do protocolo. Ela operava sobre os protocolos UDP e IP ou IPX. A segunda versão do SNMP (SNMPv2) revisou a primeira, incluindo funcionalidades nas áreas de desempenho, segurança, confidencialidade e gerenciamento das comunicações. A terceira versão do SNMP (SNMPv3) não introduziu mudanças no protocolo, ela simplesmente adicionou canais seguros via criptografia e algumas novas convenções, conceitos e terminologias. A incorporação de canais seguros foi bastante importante, pois um dos grandes problemas do SNMP desde sua primeira versão era a segurança do protocolo.

O protocolo SNMP continua sendo importante no controle da rede e seus ativos. Embora não auxilie na solução do problema em estudo, ele é muito utilizado para verificar o correto funcionamento da rede.

2.3 Ferramentas Tradicionais de Análise de Rede

O processo de detecção de problemas em rede utiliza normalmente algumas ferramentas simples, mas que auxiliam no mapeamento da maioria dos pequenos problemas enfrentados pelos usuários. Falhas em dispositivos físicos, erros em portas lógicas e problemas de conectividade com a Internet ou sistemas, podem ser solucionados, na maioria das vezes, com essas ferramentas. Geralmente, quando um dispositivo está sofrendo falhas, como uma placa de rede perdendo pacotes ou erro no roteamento, é possível usar comandos como `ping` e `traceroute` para detectar os erros rapidamente.

Existem outras ferramentas que auxiliam o administrador de rede que não serão discutidas, mas que tem importância em seus cenários específicos. Nesta seção serão avaliadas algumas ferramentas utilizadas no processo de solução de problemas de rede (`ping`, `traceroute` e `nmap`) e uma ferramenta (Nagios) que utiliza o protocolo SNMP.

2.3.1 traceroute

O comando `traceroute` (e seu correlato: `tcptraceroute`) é utilizado para mostrar o caminho na rede percorrido por um pacote, mostrando os roteadores alcançados e o tempo gasto em cada salto (*hop*). Com ele é possível saber o número de saltos realizados e o tempo gasto em cada um deles entre o dispositivo do usuário e o servidor destino.

Esta ferramenta é útil para garantir que as rotas entre um dispositivo cliente e o provedor de um serviço estejam corretas. Se durante uma requisição uma rota equivocada é informada, o usuário pode ter problemas no acesso realizado. A análise acima só é correta e precisa quando a requisição de um usuário é atendida por apenas um único

servidor. Quando mais servidores são utilizados para balancear a carga de um serviço, o `traceroute` pode trazer tantas rotas distintas quanto a quantidade de servidores do balanceamento. Isso torna o `traceroute` inefetivo, pois não existem garantias de que o problema enfrentado por uma requisição será enfrentado por outra, no acesso ao mesmo serviço.

2.3.2 ping

O comando `ping` envia pacotes ICMP de uma origem para um destino na rede, contabilizando o tempo de ida do pacote (origem $\leftrightarrow$ destino) e o tempo de volta, calculando informações como o tempo médio de cada requisição, quantidade de pacotes perdidos, dentre outros. Essas informações auxiliam o administrador de rede no mapeamento de pontos que não estão alcançáveis (sem retorno de resposta) ou então um momento de gargalo na rede (em que as respostas possuem tempos altos dado um tempo considerado padrão). Um dos maiores problemas na utilização do `ping` é que os sistemas operacionais modernos geralmente vêm com o firewall ativo para bloquear o tráfego ICMP.

2.3.3 nmap

As ferramentas anteriores são utilizadas para determinar o tempo de resposta e o caminho percorrido em uma requisição. A ferramenta `nmap`, diferente das outras duas citadas, auxilia o administrador de rede na coleta de mais informações sobre a rede. Ao invés de capturar informações de tempo de resposta ICMP ou o caminho percorrido por uma requisição, ela rastreia portas abertas, serviços ativos e até vulnerabilidades utilizando sua base de assinaturas de serviços. O `nmap` também é capaz de utilizar essas assinaturas para detectar os softwares utilizados por serviços em portas bem definidas, a versão desses softwares e o sistema operacional que está em execução. Infelizmente esta ferramenta é limitada a informações do tipo ($host \leftrightarrow host$) e não fornece mecanismos para relacionar os fluxos entre as aplicações ou mapear o caminho de requisições.

2.3.4 Nagios

O Nagios, assim como outras ferramentas que utilizam o SNMP como protocolo [16, 18, 29], é uma ferramenta capaz de armazenar o mapeamento estrutural de uma rede, detectar quando ativos de rede perdem conectividade ou serviços ficam fora do ar. Embora ele utilize SNMP para coletar e até modificar informações em dispositivos de rede com suporte a SNMP, ele não se limita somente a isso. É possível instalar agentes do Nagios nos servidores para uma coleta mais fina e controle granular de aplicações (por exemplo, do processamento de um banco de dados executando em um servidor).

O Nagios é uma ferramenta que auxilia o administrador de rede, mas não consegue associar problemas a requisições realizadas por clientes. Embora seja uma ferramenta muito útil, ele não consegue extrair as informações necessárias para resolver os novos problemas

encontrados em *data centers* modernos. Nos próximos capítulos serão estudadas ferramentas mais complexas e capazes de determinar uma gama maior de problemas em ambientes distribuídos.

2.4 Considerações Finais

Neste capítulo foram abordadas técnicas de programação utilizadas na construção de serviços e aplicações de rede e algumas ferramentas tradicionais utilizadas na análise de anomalias. A compreensão de como as aplicações são desenvolvidas é importante para entender a complexidade de uma requisição e o que é necessário para ser capaz de mapeá-la. Já o estudo das ferramentas serve de base para entender por que elas são insuficientes nos *data centers* atuais. Nos próximos capítulos serão avaliadas ferramentas capazes de detectar anomalias usando vários conceitos explorados neste capítulo.

Capítulo 3

Métodos Intrusivos de Detecção de Anomalias

Neste capítulo são apresentados os principais trabalhos para detecção de anomalias que possuem características intrusivas. O estudo desses trabalhos é importante para compreender por que eles não atendem as características esperadas de uma ferramenta genérica de detecção de anomalias. No capítulo anterior foram abordadas metodologias de programação de serviços de rede e algumas ferramentas utilizadas na detecção e resolução de problemas. Conforme os ambientes se tornam mais complexos, a detecção de problemas e suas soluções também. Os métodos intrusivos são assim nomeados pois, para detectar anomalias, precisam modificar a forma de coleta de informações das aplicações, o código dos protocolos ou *frameworks* utilizados no desenvolvimento das ferramentas. Essas modificações facilitam a reconstrução dos caminhos de uma requisição, além de permitir a fácil detecção de problemas. Por outro lado, elas limitam o escopo de atuação da ferramenta.

Um dos problemas da abordagem intrusiva é que sua aplicação fica limitada à arquitetura desenvolvida. Algumas propostas [4, 33] criam modelos bem definidos que são utilizados para mapear as aplicações, enquanto outras [7, 39] modificam as aplicações e seus protocolos para transportar informações (como, por exemplo, identificadores de conexões), com o intuito de facilitar a reconstrução do caminho de uma requisição. Cada uma destas abordagens agrega informações, mas fica limitada somente ao seu escopo de aplicação.

3.1 Magpie

O Magpie, desenvolvido em [4], tem como objetivo identificar os caminhos de controle e os recursos consumidos de cada requisição entre clientes e servidores. Para atingir seus objetivos, o Magpie monitora e analisa eventos gerados pelo núcleo (*kernel*) do sistema operacional, por *middlewares* e por componentes das aplicações. Diferente de outras abordagens, e.g. [7], Magpie não utiliza um identificador único (UID) para rastrear o caminho completo de uma requisição. Ao invés disso, ele utiliza esquemas previamente definidos

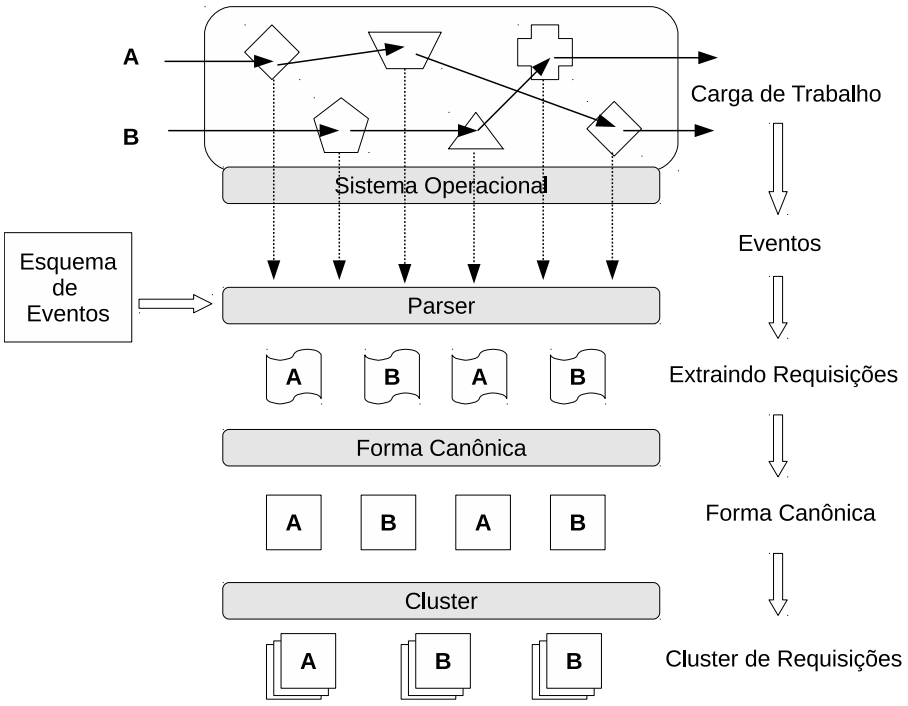


Figura 3.1: *Fluxo de execução do Magpie [4]. As requisições são coletadas dos componentes das aplicações e transformadas em eventos. Depois um* parser*, utilizando um esquema previamente fornecido, extraí as requisições que são canonizadas e agrupadas em clusters.*

pelos desenvolvedores das aplicações. Os esquemas indicam como os eventos do sistema estão relacionados e quais atributos são utilizados para se construir os relacionamentos entre os eventos e as requisições.

Além dos eventos, o Magpie extrai informações do consumo de recursos e do caminho de controle de cada requisição. Ele é capaz de agregar as informações de consumo de CPU, disco e rede e associá-las às requisições apropriadas. As associações são feitas enquanto as requisições ainda estão ativas usando o esquema pré-definido e levando-se em consideração a proximidade temporal dos eventos. Com as associações definidas, é possível criar modelos compactos das requisições para compará-los com os esquemas pré-definidos, ou seja, o comportamento esperado das aplicações. A Figura 3.1 ilustra as fases e processos utilizados pelo Magpie para identificar e coletar os eventos gerados pelas requisições.

As interações entre os processos são extraídas por meio da análise de várias entradas de comunicação presentes nos fluxos de eventos. Os fluxos são extraídos dos eventos de rede coletados enquanto as aplicações se comunicam. Dentre esses fluxos, é realizada uma busca por conjuntos que possam ser agregados de tal forma que seja possível extrair um conjunto representativo de eventos. Esse conjunto é utilizado na construção de um modelo mais compacto para auxiliar nas análises de novos conjuntos de dados.

Uma das motivações para a construção do Magpie é que as metodologias tradicionais

utilizam *logs* de atividades ou contadores que não são capazes de detectar sobrecargas em sistemas. O Magpie é mais inteligente, pois analisa e armazena informações dos recursos utilizados pelas requisições em tempo real. Para conseguir relacionar esses dados, ele utiliza o sistema de eventos ETW (*Event Tracing for Windows*) do Windows.

O traçador de eventos (ETW) grava os *timestamps* (marcas de tempo), um identificador de eventos e zero ou mais atributos. Esses atributos podem ter o mesmo conjunto de informações desde que tenham o mesmo identificador de requisição. Os resultados produzidos pelos eventos são ordenados pelo horário em que acontecem e as informações são extraídas dos *logs* de eventos com a utilização de um analisador (*parser*) de requisições. Uma vez determinados quais eventos pertencem a quais requisições, é construído um modelo descritivo das requisições que captura o fluxo de informações de controle e a utilização dos recursos do ambiente.

Para relacionar os vários tipos de fluxo são utilizados esquemas (mapeamentos de como eventos acontecem em um aplicativo) que especificam como cada atributo está relacionado a um evento. Os eventos acontecem ao longo do tempo, por isso o Magpie define um intervalo válido em que eventos podem ser agregados. Uma vez terminado o intervalo, não é possível realizar nenhuma nova agregação.

Os eventos não estão limitados apenas a acontecimentos na rede, mas ao ambiente como um todo. Outros exemplos de eventos são: uso de CPU, acesso a disco, uso de memória, troca de contextos, ciclos de CPU usados por uma *thread*, dentre outros. Assim, quando estes eventos são vinculados a uma requisição, isso indica que uma quantidade relevante de recursos foi consumida naquela requisição.

Uma característica importante do Magpie é que os esquemas precisam ser criados por pessoas que entendam as limitações da aplicação, como o sistema analisado funciona e como os recursos são consumidos. Para algumas aplicações mais simples, como requisições WEB e DNS, esquemas padrão de mapeamento de eventos se comportam bem, mas para sistemas mais complexos são necessários esquemas específicos.

Quando as requisições causam atividades em muitas máquinas na rede, é necessário agregar os eventos coletados de cada uma delas. Como cada máquina possui seu próprio relógio e os *timestamps* das mensagens são baseados nos relógios locais, é necessário manter em execução um *parser* de requisições *online* em cada máquina, canonizar os fragmentos de requisição e então executar uma ferramenta *offline* para combinar as requisições canonizadas conectando eventos de sincronização entre os pacotes enviados e recebidos.

A sincronização entre as várias máquinas da rede utiliza primitivas implícitas do sistema operacional como, por exemplo, chamadas de `send` e `receive`. Além disso, Magpie anota esses eventos usando eventos sintéticos (*Signal, Wait* e *Resume*) inseridos no analisador de requisições. Usando os eventos sintéticos é possível visualizar a causalidade entre eventos. Por exemplo, se existir uma causalidade entre eventos do tipo *Signal* e *Resume*, eles podem estar relacionados por alguma variável ou atributo compartilhado do sistema operacional que será utilizado para vinculá-los. Esse processo de extração de causalidade é realizado pela ferramenta *offline* do Magpie.

Em resumo, o Magpie pode ser considerado como uma boa solução para detecção de

gargalos de rede. Entretanto, os seus principais problemas são a necessidade de pessoal qualificado para construir os esquemas das aplicações, o trabalho constante de atualização dos esquemas, a forte dependência do ETW do Windows e a fase *offline* para análise dos eventos.

3.2 Pinpoint

O Pinpoint [7] analisa falhas em sistemas distribuídos utilizando os caminhos das requisições como fonte de informações. Com essas informações, são geradas macro visões de como os componentes interagem entre si sem se preocupar com os componentes isoladamente. Antes da extração das interações dos componentes, são extraídos os caminhos das requisições. A extração utiliza como fonte de dados as comunicações entre os componentes de rede, começando pela requisição do usuário e passando por componentes caixa preta até que a requisição pelo serviço tenha chegado ao seu destino. Com os dados coletados, é possível aplicar técnicas estatísticas para inferir o funcionamento do sistema.

O Pinpoint segue dois princípios: a medida básica dos caminhos de comunicação e a análise estatística de comportamentos. A medida básica dos caminhos é a representação da média de um grupo de caminhos coletados de ambientes caixa preta heterogêneos com a agregação de informações locais da utilização de recursos. Já os comportamentos são extraídos do grande volume de requisições coletadas utilizando técnicas de detecção de desvios de comportamentos avaliados como normais.

O Pinpoint utiliza os dois princípios citados no gerenciamento de falhas e evolução de sistemas. O gerenciamento de falhas consiste no processo completo de detecção, diagnóstico e reparo do software ou hardware envolvidos em uma falha. A chave do processo é a definição estatística do que é um comportamento normal. A partir dessa definição é mais fácil guiar o processo de diagnóstico e determinar modificações não esperadas utilizando somente os caminhos capturados. Com isso, é possível analisar o impacto no ambiente completo, pois depois que um problema é diagnosticado é possível estimar quais caminhos foram afetados com base nos pontos que falharam. O comportamento de um sistema pode mudar com a sua evolução e os componentes que antes eram afetados podem deixar de ser. Um dos problemas da evolução de uma aplicação é que ela ocorre em estágios e com poucos testes de carga (muitas vezes insuficientes) e, por isso, a forma como os componentes interagem pode ser modificada.

Existe ainda a diferença entre evolução e modificação de comportamento. A aplicação pode evoluir e ganhar novas funcionalidades, mas não deveria perder as suas características anteriores (a não ser quando estas mudanças são esperadas). Nesse aspecto, o Pinpoint alerta quanto a mudanças de comportamento relacionando os componentes influenciados. Ele faz isso comparando o comportamento atual com o histórico da aplicação e determina quando um novo comportamento apareceu e quando um antigo desapareceu.

A definição de caminhos representa o controle do fluxo das informações, os recursos alocados e características de desempenho associadas com a requisição de um serviço. Quando existem vários caminhos, estes podem ser agrupados em sessões para facilitar a

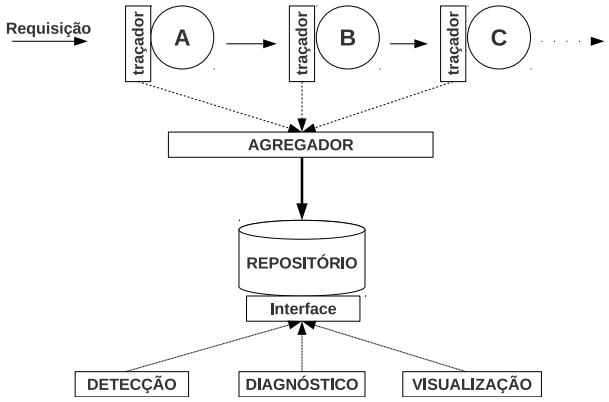


Figura 3.2: *Modelo de caminhos baseado na arquitetura de análise, ilustrando os 3 módulos utilizados no Pinpoint [7].*

interpretação e depuração das informações.

A arquitetura de análise de caminhos do Pinpoint consiste em três módulos: traçador (*tracer*), agregador e repositório de dados. A Figura 3.2 mostra os principais módulos do Pinpoint e como eles estão relacionados.

O traçador é responsável por capturar as requisições e armazenar todas as observações realizadas ao longo do caminho da requisição. Para isso, é utilizado um identificador único para cada requisição (ID) que deve ser repassado por todo o caminho da requisição. O identificador é utilizado no mapeamento do caminho completo da requisição. Para evitar modificações dos protocolos de rede para armazenar o caminho percorrido pelo identificador, são utilizados os campos de extensão dos cabeçalhos (como no HTTP ou no SOAP) ou então o armazenamento local dos identificadores das requisições. A propagação de IDs facilita a agregação de dados, pois basta unificar os caminhos que possuam o mesmo ID para que sejam extraídos corretamente os caminhos das requisições.

Para avaliar a eficácia da metodologia, os autores do Pinpoint realizaram experimentos com o servidor HTTP Jetty [42] (desenvolvido em Java) modificado para gerar um ID único para cada requisição. Como o Jetty é integrado ao JBoss e usa a mesma JVM, a Classe ThreadedLocal foi utilizada para armazenar os IDs que são propagados nas requisições. Uma vez que o ID é propagado entre as instâncias do JBoss, é mais fácil mapear o caminho percorrido pela requisição do usuário por toda a aplicação até que o serviço retorne a resposta para o usuário. Quando o servidor invoca uma EJB (*Enterprise Java Bean*) remota, o protocolo RMI (*Remote Method Invocation*) modificado repassa o ID para o EJB alvo de forma transparente. A Figura 3.3 mostra como os IDs são propagados.

O agregador de informações recebe as observações dos traçadores e unifica os caminhos utilizando os IDs gerados. Ele armazena os caminhos no repositório central e permite que filtros sejam aplicados para reduzir o volume de dados armazenado no banco de dados.

A utilização de caminhos contribui para o gerenciamento de falhas e principalmente para a detecção, diagnóstico, rápida capacidade de isolamento, análise de impacto, reparação e informação dos problemas. Essas informações são importantes pois serão utilizadas

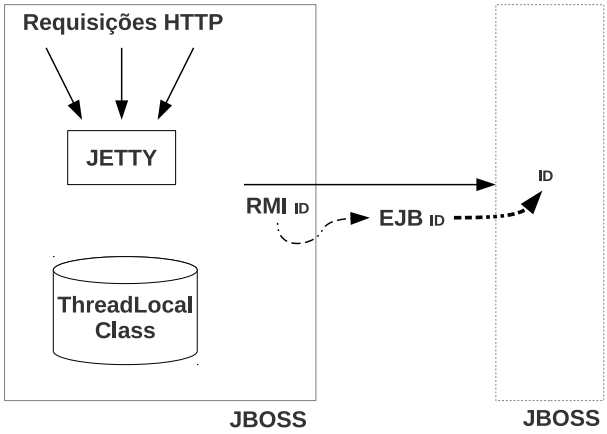


Figura 3.3: *Propagação de IDs no Pinpoint [7].*

como experiência para melhorar futuras detecções. Neste aspecto o objetivo principal do Pinpoint é a recuperação de falhas, pois as ocorrências de problemas são inevitáveis.

A detecção de falhas utilizando caminhos apresenta alguns problemas como, por exemplo, a colisão entre caminhos em requisições não finalizadas. Por isso, a coleta de dados de caminhos incompletos é crucial, pois é utilizando os sinais de interrupção da conexão e os IDs que caminhos quebrados têm seus fluxos reconectados. Um exemplo prático são requisições HTTP que são reiniciadas pelo usuário ao recarregar uma página que não terminou de ser carregada.

Outro problema são as anomalias na estrutura do caminho. Neste caso, a solução para garantir que os resultados obtidos sejam confiáveis é utilizar uma gramática livre de contexto probabilística (PCFG - *Probabilistic Context Free Grammar*), treiná-la por pelo menos cinco minutos em um ambiente livre de falhas e só então analisar como as falhas seriam detectadas. Os caminhos são considerados anômalos somente quando eles não podem ser gerados pela PCFG. Para avaliação do Pinpoint os autores fizeram testes em um *benchmark* de comércio eletrônico (*e-commerce*) chamado TPC-W WIPSo [11]. A taxa de detecção de falhas nos testes foi de 90%.

Variações de latências também podem ser consideradas como anomalias, principalmente quando ocorrem picos em trechos de um caminho. Essas variações podem significar que ao longo do caminho os componentes estão enfrentando falhas ou estão sobrecarregados.

O Pinpoint também permite a análise de múltiplos caminhos. Para isso ele utiliza duas abordagens: árvore de decisões e força bruta. Nas árvores de decisões, as folhas representam os possíveis candidatos a raiz de um problema. No caso da força bruta, são geradas todas as possíveis combinações de falhas. Em ambos os casos, as possíveis falhas são pontuadas utilizando probabilidade. Além disso, em ambas as soluções é necessário filtrar o ruído gerado por possíveis erros no modelo. Os resultados dos experimentos realizados pelos autores acertou corretamente a raiz de um problema em 93% dos casos. Todavia, enquanto a árvore de decisões gerou apenas 23% de falsos positivos, a força bruta

gerou 50%.

Os sistemas mudam com o tempo por atualização de versão, correções de falhas ou outras modificações e estas mudanças costumam trazer novas características, suporte a novos hardwares e melhoria de desempenho. O Pinpoint detecta essas mudanças utilizando o histórico armazenado de cada aplicação já analisada e alerta os administradores. Além disso, uma análise minuciosa dos caminhos pode auxiliar na compreensão das evoluções mostrando as dependências entre os componentes dos softwares, validando novas características (já esperadas) ou detectando regressões entre as ligações de componentes que deveriam estar interligados.

Uma distinção chave entre o Pinpoint e a análise estática modelada utilizando UML manualmente, por exemplo, é o fato de que os caminhos capturam as dependências reais dos componentes, enquanto a análise estática pode incluir dependências em potencial entre componentes mas que talvez não existam de fato. Embora a ideia da utilização dos caminhos tenha provado ser útil na análise de várias aplicações, ela sozinha não resolve os problemas. Por exemplo, a implementação atual do traçador não é capaz de contabilizar informações de componentes de nível mais baixo como discos em RAID ou caches *web* transparentes.

O Pinpoint foi um dos primeiros trabalhos aplicáveis às tarefas de gerenciamento de falhas. Existem outras abordagens mais recentes como o Netmedic [24] e o Sherlock [2], que serão analisadas no próximo capítulo e que também trabalham nesse sentido. Contudo, a modificação dos protocolos para o transporte de informações como o ID torna o Pinpoint intrusivo e limita seu escopo de aplicação. Por outro lado, torna a construção dos caminhos das requisições mais fácil.

A propagação de um ID para rastrear uma requisição pelo caminho causal utilizado por Pinpoint foi uma ideia inovadora. Trabalhos mais recentes como o X-Trace [13] utilizam essa mesma ideia, adicionando mais informações. No caso do X-Trace, não apenas o ID mas todo um conjunto de metadados é propagado pelo caminho causal de uma requisição.

3.3 PIP

Ferramentas como `gdb` e `gprof` são muito utilizadas na depuração em baixo nível de aplicações, mas acabam se tornando pouco eficientes com o crescimento e a complexidade das aplicações. A forma mais comum de monitoramento de comportamento de sistemas são as saídas de *logs*. Entretanto, os *logs*, em geral, armazenam poucas informações e exigem muito trabalho manual para se detectar os problemas em uma requisição.

Para auxiliar no processo de desenvolvimento e depuração de aplicações distribuídas, PIP [33] oferece um conjunto de ferramentas aos desenvolvedores para o mapeamento de problemas. Ele possui uma infraestrutura que compara o comportamento dos programas em execução com o comportamento esperado pelos desenvolvedores. Essa expectativa dos desenvolvedores é expressa em uma linguagem capaz de modelar os eventos de uma aplicação e contabilizar os recursos utilizados durante sua execução. Seu objetivo é atender três tipos de usuários: desenvolvedores de software, aprendizes do sistema e

administradores que vão monitorar mudanças.

Além das expectativas dos desenvolvedores, PIP também utiliza caminhos causais e, de forma análoga ao Pinpoint [7], utiliza um identificador para verificar o comportamento de uma aplicação. PIP oferece uma linguagem concisa para descrever comportamentos esperados de aplicações e um conjunto de ferramentas para coletar eventos, validar comportamentos coletados e visualizar os resultados obtidos. Ele também é capaz de gerar automaticamente expectativas de uma aplicação utilizando os traços coletados dela.

Para que PIP funcione, o desenvolvedor deve descrever qual o comportamento esperado da aplicação durante sua execução e modificá-la para utilizar uma biblioteca de anotações. Geralmente, aplicações distribuídas utilizam *frameworks* para facilitar o seu desenvolvimento e, por isso, é possível inserir a biblioteca de anotações diretamente no *framework* para mapear todas as comunicações e recursos utilizados a cada evento gerado pela aplicação. Com os eventos coletados, os comportamentos descritos pelo desenvolvedor serão testados para validar a aplicação. É possível que alguns comportamentos não sejam esperados e, neste caso, ao invés de invalidar uma ação, PIP alerta quanto a este novo comportamento não esperado.

A unidade mais básica de um comportamento é uma instância em um caminho. A instância é sempre causal e sempre uma resposta a uma requisição externa de um usuário. Por exemplo, em um sistema de arquivos distribuído, a instância do caminho pode ser a leitura ou a escrita de um bloco de dados. Cada instância em um caminho é ordenada pelos acontecimentos dos seus eventos. PIP define três tipos de eventos: tarefas, mensagens e notícias. As tarefas são chamadas de procedimentos e possuem um tempo esperado de início e fim e as medidas de utilização de recursos. As mensagens são quaisquer comunicações que aconteçam entre dois nós ou entre *threads*. E as notícias são os *logs* de sistema.

PIP não é uma aplicação única, ele é um sistema composto por uma gama de programas que trabalham em conjunto para coletar, verificar e mostrar o comportamento das aplicações distribuídas. A Figura 3.4 mostra como as aplicações do PIP interagem em cada etapa do seu funcionamento. O fluxograma mostrado é executado em etapas. Primeiro é feita a coleta das anotações dos eventos gerados pela aplicação nos traços locais. Os traços são coletados e conciliados em um banco de dados de caminhos e as expectativas dos desenvolvedores são validadas utilizando as informações do banco de dados. Para avaliar uma expectativa, é necessário validar ou invalidar as instâncias em um caminho individualmente e agregar as informações dessas instâncias em conjuntos.

O reconhecimento e validação de instâncias em um caminho são feitos por gramáticas que definem sequências de eventos válidos ou inválidos. Quando uma sequência de eventos não esperada ocorre, o desenvolvedor é informado de que um comportamento não esperado foi detectado. Essa violação da expectativa do desenvolvedor não quer dizer necessariamente que a aplicação tem um problema. Erros na descrição das expectativas ou nas anotações nos traços podem ocorrer algumas vezes. Após a execução de cada uma das etapas do fluxograma, o PIP fornece uma aplicação gráfica para visualizar a estrutura causal de comunicação da aplicação.

O funcionamento das ferramentas do PIP é conceitualmente parecido com dois trabal-

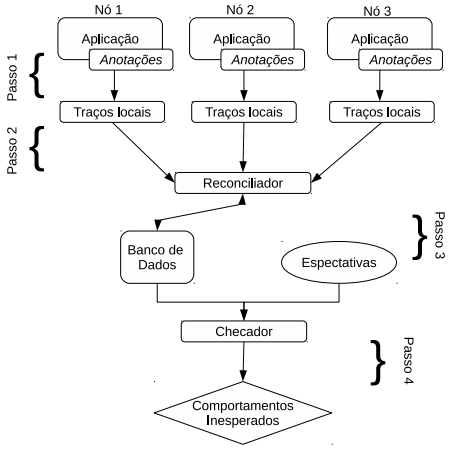


Figura 3.4: *Processo de utilização das ferramentas do PIP para depurar programas distribuídos por etapas. Passo 1: anotações nos programas. Passo 2: reconciliação e armazenamento em banco de dados. Passo 3: comparação das expectativas criadas com o comportamento capturado. Passo 4: exposição dos comportamentos não esperados detectados [33].*

hos anteriores. A utilização de um ID propagado pela biblioteca de anotações é similar a ideia do Pinpoint[7] com a propagação do ID entre as EJBs, mas com a diferença de adicionar mais informações a cada evento gerado. Assemelha-se também com o Magpie [4], pois é necessário descrever como os eventos da aplicação estão relacionados. Entretanto, o PIP é mais abrangente e consegue reduzir o problema do paralelismo dividindo este tipo de comportamento em caminhos e o paralelismo interno como *threads* de eventos dentro do caminho.

O PIP foi o primeiro sistema na época a utilizar dois conceitos diferentes ao mesmo tempo, a análise de caminhos e a validação automatizada de expectativas de uma aplicação. Como visto anteriormente, pelo menos dois trabalhos utilizam parcialmente as mesmas ideias contidas no PIP, mas não ao mesmo tempo.

3.4 X-Trace

Diferente de todas as abordagens de detecção de anomalias intrusivas, o X-Trace [13] propõe a modificação de toda a pilha de protocolos de rede. Por isso sua aplicação é considerada mais teórica do que prática, embora existam algumas validações práticas implementadas [12].

O X-Trace é capaz de auxiliar na construção dos caminhos causais ao propagar metainformações por todo o caminho de comunicação. Esta abordagem é semelhante à utilizada no Pinpoint [7], mas a propagação de metainformação não fica restrita a camada de aplicação, ela é feita desde as camadas superiores, até a camada de rede. O objetivo de se propagar metainformação em várias camadas é extrair os caminhos causais das comunicações de forma mais precisa e completa.

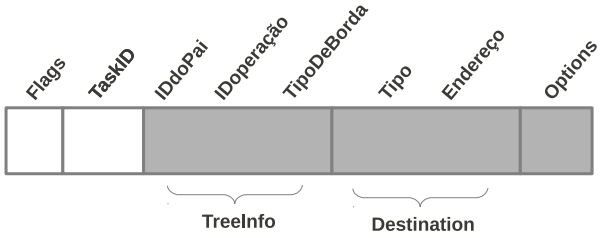


Figura 3.5: *Formato dos campos dos metadados do X-Trace [13].*

Em um ambiente distribuído, os caminhos gerados por requisições dos usuários nem sempre são os mesmos. Servidores de balanceamento de carga, de cache, de DNS, entre outros, podem não ser os mesmos quando for necessário reproduzir um problema. Por isso, é importante se determinar corretamente o caminho relacionado à requisição que gerou algum erro. Em sistemas em que o X-Trace está habilitado, administradores de diferentes domínios administrativos (DAs) podem extrair informações relacionadas ao problema ou fornecer dados para auxiliar o processo.

Cada requisição gerada é tratada como uma tarefa dentro do X-Trace e cada tarefa pode requisitar outras tarefas formando uma cadeia. Cada tarefa está associada a apenas uma camada de rede e pode gerar novas requisições para as camadas inferiores. A chave do funcionamento do X-Trace está na decisão de projeto que exige que os metadados sejam enviados junto com os dados (*in-band*), garantindo que a coleta de informações seja precisa e utilize o mesmo caminho que os dados coletados.

Um usuário que requisite informações de um caminho problemático pode receber somente os dados relativos à sua rede local. Da mesma forma, os dados capturados no DA serão enviados somente aos administradores da rede. Os dados contendo relatórios podem ser armazenados em bancos de dados locais do DA ou em ambientes remotos definidos no campo opcional na estrutura dos metadados.

Para que seja possível mapear os metadados do X-Trace entre as várias camadas, é necessário modificar os protocolos para inserir a estrutura do metadado dentro do protocolo. Por isso, sempre que possível, os metadados são inseridos dentro dos campos opcionais dos protocolos.

O formato básico dos metadados (Figura 3.5) consiste em dois campos obrigatórios e três opcionais. O campo obrigatório `Flags` especifica quais dos três campos opcionais estão disponíveis. O `TaskID` possui um ID único e a janela de tempo em que este ID é considerado válido. O `TreeInfo` possui informações sobre o caminho. O `Destination` armazena os dados de rede para onde o X-Trace deve enviar os relatórios e o campo `Options` está reservado para acomodar novas funcionalidades.

A utilização de metadados torna o mapeamento do caminho entre os equipamentos de rede mais fácil. Os mapeamentos carregam três informações entre as tarefas: IDdoPai, IDoperacao, TipoDeBorda. Essas informações são vinculadas ao campo opcional TreeInfo e registram os relacionamentos causais entre as operações. O IDdoPai é o ID repassado pela tarefa pai aos seus filhos, o IDoperação é um ID único gerado aleatoriamente (e com

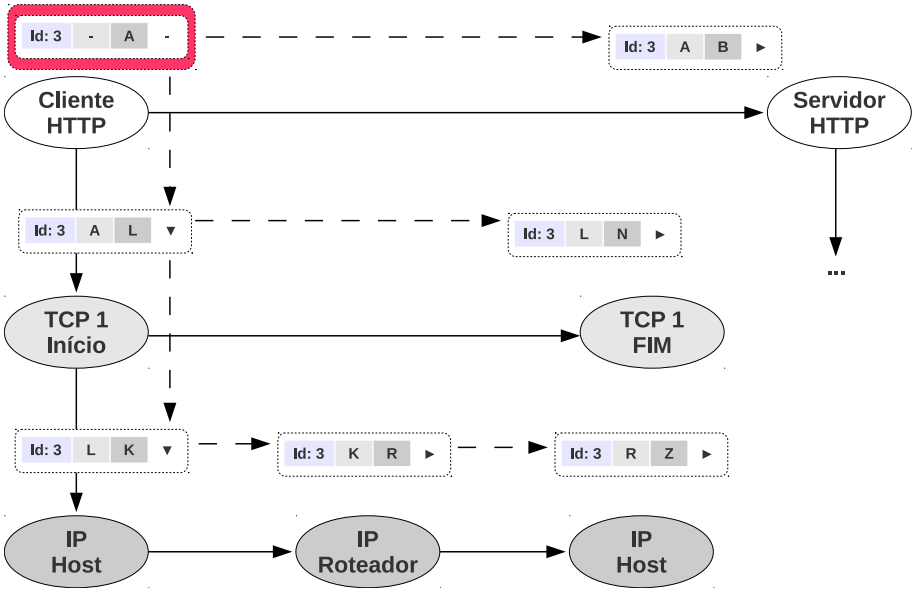


Figura 3.6: *Exemplo de propagação dos metadados do X-Trace ao longo das múltiplas camadas de rede. Os objetos acima de cada ação do host representam as informações de metadados do X-Trace naquele momento, em que os valores são: ID da requisição, IDdoPai, IDoperacao e TipoDeBorda, respectivamente. A seta para baixo invoca o método* `pushDown()` *e a seta para direita invoca o método* `pushNext()`*. [13]*

um bom intervalo de tempo entre as repetições) e o campo TipoDeBorda indica o tipo de conexão entre dois nós adjacentes (na mesma camada ou em camadas distintas). A Figura 3.6 ilustra a propagação de metadados entre as várias camadas.

Para construir o grafo de relacionamentos utilizando os metadados é necessário detectar quando uma tarefa é executada na mesma camada ou em uma camada inferior. Por isso a propagação de metadados utiliza duas funções diferentes:

- `pushDown()`: propaga os metadados para uma camada inferior;
- `pushNext()`: propaga os metadados para uma requisição na mesma camada.

Os metadados propagados entre as camadas são utilizados na reconstrução do caminho causal da requisição. A reconstrução da árvore de tarefas é dividida em duas fases. Enquanto os dados dos traços de rede são recolhidos, em cada nó em que for detectada a existência de metadados do X-Trace será gerado um relatório (este relatório é gerado em qualquer uma das camadas). O relatório contém o horário local, o ID da tarefa referenciada e informações do nó que está enviando o relatório. Depois é feita a reconstrução *offline* da árvore de tarefas examinando as arestas *down* e *next* (a aresta direcionada gerada com as chamadas das funções `pushDown()` e `pushNext()`).

A geração de relatórios e a visibilidade do caminho da requisição que trafega por um DA fica visível somente a ele. Todavia, quando existe uma configuração para enviar os relatórios para um banco de dados remoto ou servidor público de traços, os dados podem

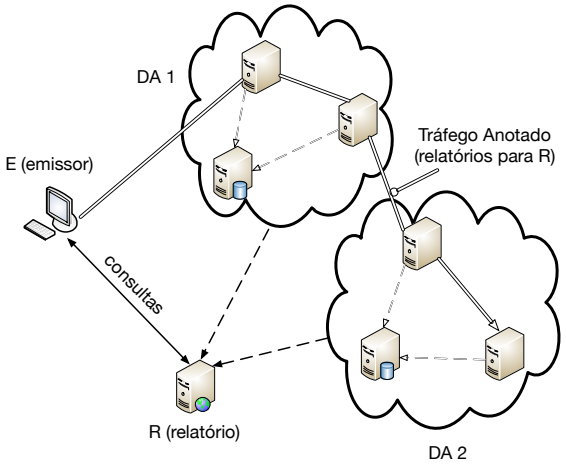


Figura 3.7: *O tráfego gerado pelo emissor E é capturado ao longo dos DAs com que ele tem contato e, no caso ilustrado, foi disponibilizado para acesso aos relatórios no servidor R [13].*

ficar acessíveis também aos usuários. Na Figura 3.7 está um exemplo de como o usuário poderia analisar o caminho gerado pelo X-Trace mesmo que a requisição realizada tenha trafegado por diferentes DAs antes de retornar.

O X-Trace não modifica apenas os protocolos, ele altera também os sistemas. Isso deve ser feito para informar quando novas tarefas são criadas ou na mudança entre as camadas de rede. A modificação de protocolos e aplicações envolve três etapas: incluir os metadados do X-Trace nas trocas de mensagem, incluir a lógica de propagação dos metadados na implementação seguindo os caminhos causais e, opcionalmente, incluir chamadas adicionais em pontos de interesse.

O suporte ao X-Trace pela camada de rede ou sistema precisa embutir os metadados nas mensagens trocadas. A dificuldade da implementação depende, geralmente, da especificação original do protocolo. Quando existe a possibilidade de extensões, o trabalho é mais fácil. De outra forma, é necessário modificar as implementações. Para que o X-Trace funcione, as aplicações e protocolos devem oferecer suporte, em especial, a dois aspectos de propagação do identificador: é necessário propagar a informação dos metadados à saída correta e manipular os metadados com as funções `pushDown()` e `pushNext()` para que correspondam com a relação causal das comunicações. A Figura 3.8 ilustra o uso de X-Trace para o mapeamento de uma requisição de uma página WEB que efetua uma resolução de nomes via DNS.

Em [12], Fonseca *et al* analisa a implementação da propagação de metadados em outros protocolos como o 802.1X, LDAP e na réplica seletiva de DNS. Os experimentos realizados pelos autores foram realizados na infraestrutura CoralCDN [10] e no serviço *anycast* OASIS [31]. Foi possível detectar problemas como expiração do tempo de comunicação

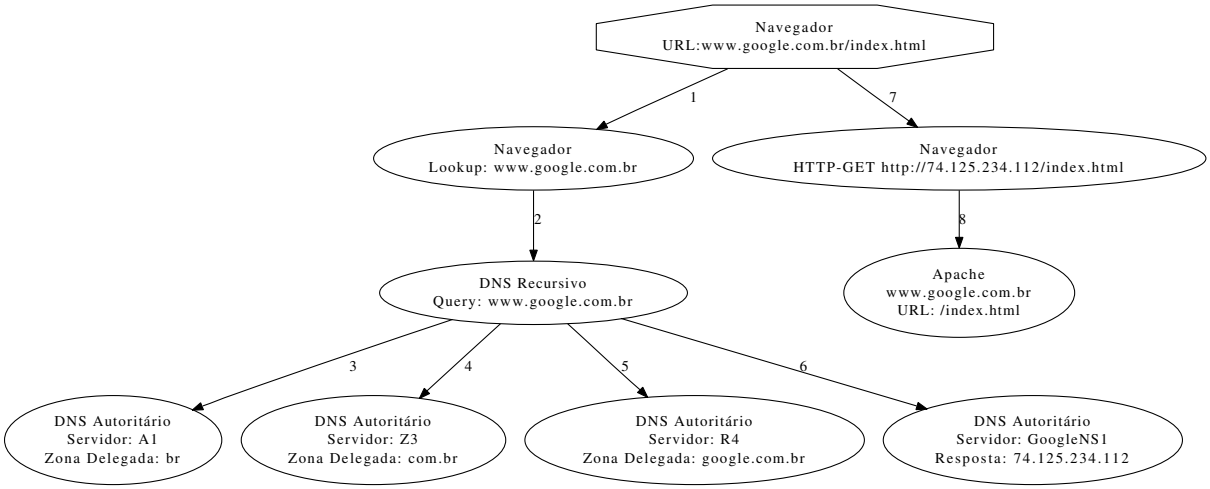


Figura 3.8: *Caminho de uma requisição HTTP que foi anotada pelo X-Trace, com relação à resolução de DNS de uma página WEB.*

por falhas de configurações, perda de pacotes (no caso do RADIUS), sobrecarga do serviço RADIUS ou LDAP e falhas de configurações do firewall. Portando, X-Trace, muito embora seja uma abordagem extremamente intrusiva, sua aplicação prática é possível como comprovado pelos experimentos realizados.

3.5 vPath

Uma abordagem menos intrusiva no quesito sistema operacional é dada pelo vPath [39]. Ao invés de implementar as modificações diretamente no sistema operacional, o vPath explora ambientes virtualizados para capturar de forma precisa os caminhos de comunicação. As modificações realizadas no vPath são incluídas no gerenciador de máquinas virtuais XEN e são transparentes para as máquinas virtuais (VMs).

O vPath considera apenas o modelo de programação síncrona de serviços de rede. Além disso, o vPath supõe que todas as requisições usam *sockets* TCP bloqueantes, ou seja, uma nova conexão só pode ocorrer quando uma anterior terminar. Essa característica bloqueante é determinante no funcionamento do vPath.

Um serviço programado com *threads* e *sockets* TCP bloqueantes permite facilmente a inferência de causalidade. Por exemplo, se um componente X envia a mensagem A para o componente Y, é gerada como resposta de Y a mensagem A'. Utilizando as informações de $(IP_{origem}, Porta_{origem}) \rightarrow (IP_{destino}, Porta_{destino})$ é possível mapear a comunicação entre os componentes $(X \rightarrow Y)$ por meio de suas mensagens (A e A'). A reconstrução da requisição utiliza as informações das conexões TCP nas chamadas de sistema (`send` e `receive`) coletadas do sistema operacional.

A construção do caminho de requisições feita pelo vPath segue uma lógica simples. Um exemplo prático está na Figura 3.9, em que uma requisição A é realizada ao Componente1.

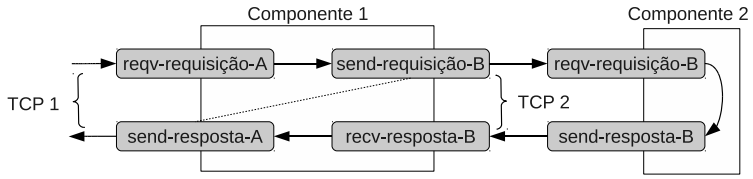


Figura 3.9: *Caminho de processamento de requisições, onde dois componentes estão se comunicando e cada requisição ou processamento é representado pelas chamadas de função* `recv` *ou* `send` *[39].*

As chamadas de sistema aparecem na figura e mapeiam as interações entre os vários componentes até que a requisição retorne. Por questões de simplificação, o vPath assume que a *thread* que enviou a requisição `send-requisição-B` e a resposta `send-resposta-A` é a mesma. Embora outras requisições possam ser feitas por *threads* diferentes.

O vPath foi implementado no ambiente de virtualização Xen. Foram feitas modificações específicas no VMM (*Virtual Machine Manager*) do Xen para capturar informações das *threads* e das trocas de mensagens. Essas modificações foram feitas no VMM para que não fosse necessário modificar as máquinas virtuais. Dessa forma, o vPath extrai o caminho completo de requisições de qualquer sistema operacional virtualizado com poucas alterações.

O vPath é composto pelo monitor online e analisador de *logs offline*. O monitor online gera *logs* continuamente sobre as chamadas de sistemas `send` e `recv` e o analisador *offline* analisa os *logs* gerados para descobrir os caminhos de requisição, processamento e características de desempenho. Além disso, o analisador *online* reconstrói as requisições utilizando dois tipos de causalidade: a causalidade Intra nó (mensagem recebida A gera a mensagem de saída B) e a causalidade Inter nó (mensagem A enviada por um componente corresponde à mensagem A' recebida em outro). Para inferir corretamente a sequência de acontecimentos, o analisador *online* rastreia e monitora as *threads* de um serviço usando informações extraídas de conexões TCP como $<IP_{origem}$, $PORTA_{origem}$, $IP_{destino}$,$PORTA_{destino}>$. Isso permite identificar corretamente qualquer conexão TCP em qualquer momento e de forma única.

O analisador *online* foi implementado diretamente no VMM do Xen. A implementação teve de lidar com problemas na forma como o VMM captura as interrupções das máquinas virtuais. Além disso, quando um processo é executado dentro de uma máquina virtual, as interrupções geradas podem ser capturadas pelo VMM, ao contrário do que acontece com as *threads*, pois estas não geram interrupções da mesma forma que os processos e ficam invisíveis para o VMM.

Para coletar as informações de cada *thread* dentro das máquinas virtuais, é necessário interceptar as chamadas de sistema armazenando o DomainID da máquina virtual, o conteúdo do registrador CR3 e o conteúdo do registrador EBP. Utilizando essas informações (DomainID/CR3/EBP) é possível identificar uma *thread* dentro do processo e, consequentemente, dentro da máquina virtual correspondente. Na prática, o DomainID identifica a VM, o registrador CR3 identifica o processo dentro da VM e o conteúdo do

EBP identifica a *thread* dentro do processo.

A captura das chamadas de sistema precisa de um manipulador de chamadas de sistema especial que se registre na IDT (*Interruptor Descriptor Table*) gerada para cada VM em execução no XEN. Com a utilização do manipulador, o acesso direto realizado pela VM é desabilitado e passa a ser filtrado pelo manipulador. Com essa modificação, o vPath analisa o tipo de chamada e gera um registro de *log* se a operação for de envio ou recebimento. A interceptação das chamadas do sistema não causa grande impacto, pois as chamadas de sistema em si já realizam uma troca de contexto entre o modo usuário e o modo kernel, o que torna o custo da interceptação insignificante.

A abordagem do vPath na detecção de anomalias é bastante interessante e precisa (acertou corretamente 100% dos caminhos realizados em uma requisição de acordo com os testes realizados pelos autores), mas limitada a máquinas virtuais bem definidas. Sua contribuição mais importante é a abordagem diferente do problema de detecção de anomalias, pois ao invés de instrumentar aplicações, o vPath explora os padrões de programação de serviços de rede. Entretanto, mesmo que as máquinas virtuais não sejam modificadas, a solução do vPath fica limitada ao ambiente virtual e não pode explorar máquinas reais distribuídas na rede. Outra característica limitante é o uso de conexões bloqueantes, pois as aplicações distribuídas cada vez mais exploram o assincronismo de operações.

3.6 Comparação Qualitativa dos Métodos Intrusivos

Os métodos intrusivos estudados nas seções anteriores podem ser comparados utilizando-se algumas métricas. Para ser o mais abrangente possível será utilizada uma métrica quantitativa (com resultados numéricos extraídos diretamente dos trabalhos originais) e duas métricas qualitativas. A métrica quantitativa se refere à eficiência e as métricas qualitativas se referem à generalidade do método e sua transparência. Para que o significado destes termos fique mais claro, suas definições seguem abaixo:

- Eficiência: reflete a sobrecarga ou vazão que uma abordagem adiciona a uma aplicação ou sistema;
- Generalidade: está relacionada com configurações de hardware e software em que uma abordagem é aplicável, incluindo fatores como linguagens de programação, sincronização de relógio, presença ou ausência de *logs* de sistema, modelo de paralelismo, etc;
- Transparência: representa a habilidade de uma abordagem modificar ou não um sistema em nível de usuário ou kernel.

Uma vez que os parâmetros de comparação estão bem definidos, a Tabela 3.1 compara os cinco métodos intrusivos abordados neste trabalho. Esta tabela não cobre todos as características de todas as ferramentas e métodos.

Tabela 3.1: *Tabela qualitativa dos métodos intrusivos avaliados neste trabalho.*

	Eficiência	**Generalidade**	**Transparência**
Magpie	redução $< 4\%$	esquema de aplicações	coleta de logs do ETW e chamadas de sistema
Pinpoint	redução $< 1\%$	propagação de ID	modificação do JBoss, invisível à aplicação
X-Trace	*throughput* 15% menor	propagação de metadado	modificação de protocolos e sistemas
vPath	*throughput* 6% menor	*threads* bloqueantes	XEN modificado invisível às máquinas virtuais
PIP	3400 caminhos/s	propagação de ID	biblioteca de anotações de caminho

Além das outras informações qualitativas, a Tabela 3.1 mostra que a eficiência nem sempre pode ser medida com a redução do potencial de processamento (uso do processador pela aplicação). No caso do X-Trace e vPath, a eficiência está relacionada à vazão de dados (*throughput*) antes e depois da abordagem. Já o PIP não é a implementação de uma ferramenta, por isso, sua eficiência é medida pela quantidade de caminhos que podem ser analisados utilizando sua biblioteca de anotações em um único computador.

3.7 Outros Métodos Intrusivos

Os métodos intrusivos apresentados neste capítulo fornecem uma visão geral do problema e das diferentes técnicas que podem ser utilizadas para resolvê-lo. Além dos métodos já detalhados, há vários outros na literatura que atacam o mesmo problema. As subseções a seguir apresentam métodos mais recentes que também foram estudados ao longo deste trabalho, mas que são simples variações dos já detalhados neste capítulo.

3.7.1 Whodunit

O Whodunit [6] é, em poucas palavras, um depurador transacional para aplicações multicamada. As aplicações multicamada são muito comuns em servidores de conteúdo dinâmico na Internet. Além disso, esses servidores costumam estar em ambientes balanceados com vários servidores de aplicações e banco de dados. Como as aplicações dependem de vários outros serviços em servidores distintos, a complexidade de depuração cresce. O Whodunit consegue, além de depurar as requisições nas mais diversas camadas, contabilizar informações, como por exemplo, a interferência de travas de contenção de acesso à memória no sistema operacional, requisições de um servidor web que causaram gargalo no banco de dados, dentre outros.

As principais contribuições do Whodunit são: um algoritmo para detecção e acompanhamento de fluxos de transações em memória compartilhada (*shared memory*), um mecanismo para acompanhar a execução de transações em eventos ou estágios (principalmente com SEDA), a capacidade de medição e detecção de interferências entre transações concorrentes bloqueadas por contenção e um mecanismo para acompanhar processos em

várias máquinas. Este trabalho deve ser comparado ao Magpie [4] e PIP [33] para melhor compreensão.

3.7.2 BorderPatrol

O BorderPatrol [26] aborda a extração de informações contidas em traços de rede de forma diferente das metodologias intrusivas tradicionais. Ao invés de instrumentar diretamente as aplicações para extrair precisamente os caminhos de uma requisição como feito pelo Pinpoint [7] e Pip [33], o BorderPatrol utiliza processadores de protocolo. Quando uma aplicação dispara vários módulos (*threads* ou processos), ou quando multiplexa várias requisições, o caminho das requisições não segue, necessariamente, um modelo de controle de fluxo. Processos podem se comunicar usando diversas abstrações do sistema operacional para gerenciar concorrência. Por isso, o BorderPatrol utiliza os processadores de protocolo para mapear as requisições dentro do sistema operacional e separar múltiplas mensagens em canais únicos, permitindo a reconstrução de uma requisição. BorderPatrol, mesmo sendo utilizado em análise de sistemas caixa preta, é intrusivo, pois instrumenta o kernel e realiza captura de pacotes por interposição.

O BorderPatrol contribui com uma técnica de isolamento ativo de sistemas caixa preta, em que as requisições podem ser analisadas precisamente sem, necessariamente, interferir na habilidade de multiplexação de uma aplicação. É capaz de isolar acontecimentos, desmembrando eventos de entrada concorrentes para posterior observação do comportamento de um módulo, além de identificar e cruzar mensagens que representam pares de requisições e respostas. Ao contrário de outras abordagens, o BorderPatrol prima pela precisão ao invés de sacrificá-la por informações estatísticas.

3.7.3 Macroscope

O Macroscope [27] é uma ferramenta, que, assim como as ferramentas não intrusivas como Sherlock [2] e Orion [8], extrai dependências das aplicações de rede automaticamente. Por outro lado, o Macroscope critica o Sherlock e o Orion [8] por utilizarem somente traços de rede. A principal crítica é direcionada ao modelo de detecção de anomalias usando metodologias não intrusivas, pois apenas com informações de traços de rede, uma aplicação de detecção de anomalias seria muito limitada. Por exemplo, serviços que ocorrem com pouca frequência ou são muito dinâmicos seriam perdidos, além dos problemas relacionados à correlação de serviços independentes criando a ilusão de causalidades. Estes problemas acabam gerando muitos falsos positivos em ferramentas apenas não intrusivas.

A abordagem do Macroscope extrai dependências automaticamente e, ao contrário das metodologias não intrusivas, realiza alguma instrumentação nos dispositivos clientes. Além disso, ele unifica informações de amostras das tabelas de conexões TCP/UDP com informações dos pacotes coletados nos traços de rede. Isso permite que fluxos de rede individuais sejam associados com informações geradas por aplicações. Desta forma, o Macroscope contribui com um novo modelo de identificação de dependências de serviços que são aderentes às aplicações. Usando estas informações, o Macroscope constrói um

Grafo de Dependência de Serviços (GDS) contendo múltiplos níveis de granularidade, um ponto positivo em relação aos modelos adotados pelo Sherlock e Orion.

3.7.4 Dapper

O Dapper [37] é uma ferramenta criada para analisar problemas de desempenho nos sistemas em produção do Google. Seu projeto compartilha conceitos similares com o Magpie [4], o Pinpoint [7] e o X-Trace [13]. Como o Dapper é um estudo de caso de uma implementação real, seus resultados são focados nas necessidades e sistemas do Google. Uma ferramenta como essa deve ser tão leve e rápida que seu uso, mesmo em sistemas muito sensíveis a qualquer latência adicional (somando-se o fato de que os usuários, no geral, são impacientes com respostas lentas de sistemas), deve ser imperceptível. Por isso, o Dapper utiliza um esquema de amostras em que apenas um pequeno conjunto delas carrega informações suficientes sobre os usos mais comuns de uma aplicação. Além disso, apenas um pequeno conjunto de bibliotecas é instrumentado para coletar dados. Isto permite um nível muito bom de transparência em nível de aplicação, agregada à baixa utilização de recursos. Dessa forma, o Dapper atende a três objetivos principais: baixa sobrecarga, transparência e escalabilidade. Ele também serve como ferramenta de detecção rápida de problemas de desempenho para desenvolvedores.

3.8 Considerações Finais

Este capítulo apresentou algumas técnicas intrusivas utilizadas na detecção de anomalias em ambientes distribuídos. As técnicas descritas servem de arcabouço para compreender como as principais ferramentas intrusivas funcionam em contraponto às não intrusivas.

A compreensão do que é intrusão em um sistema e como é o seu comportamento é importante para saber o que torna ou não uma ferramenta intrusiva. Muitas vezes o limiar entre uma ferramenta intrusiva e não intrusiva é muito tênue. Por isso, este capítulo ajuda consolidar essa compreensão. No próximo capítulo serão abordadas técnicas não intrusivas, que são a base da ferramenta desenvolvida neste trabalho.

Capítulo 4

Métodos Não Intrusivos

No capítulo anterior foram apresentadas as técnicas intrusivas de detecção de anomalias e suas principais características. O modelo intrusivo de detecção de anomalias é limitado ao ambiente de análise e depende de modificações nos sistemas. Neste capítulo são apresentados os principais trabalhos para detecção de anomalias que são não intrusivos ao ambiente.

A não intrusão reflete a característica de uma abordagem não modificar o ambiente para detectar uma anomalia. Isso significa que são utilizadas apenas informações coletadas do ambiente computacional no processo de detecção. Uma ferramenta pode coletar pacotes em uma placa de rede ou por uma porta de um *switch* e, em ambos os casos, o ambiente continua intacto. Por isso, essas abordagens são não intrusivas.

Por outro lado, as abordagens não intrusivas são menos precisas que as abordagens intrusivas. Isso ocorre, pois a detecção usando apenas informações coletadas do ambiente depende de estimativas. Por isso, podem sofrer interferência externa com ruídos (dados ou informações truncadas, sobrecarga de servidores, etc.), além dos problemas inerentes ao modelo, como a baixa ocorrência de tráfego gerada por requisições esporádicas ou tráfegos constantes gerados por *downloads*.

Mesmo com os problemas inerentes aos métodos não intrusivos de detecção de anomalias, eles são melhores do ponto de vista prático. Ferramentas não intrusivas não modificam o ambiente, não descartam sistemas legados, não dependem de informações prévias de aplicações e, por isso, são mais adaptáveis em *data centers* distintos. Essas são características importantes para uma ferramenta genérica de detecção de anomalias em ambientes distribuídos e, por isso, este trabalho é baseado nos modelos não intrusivos. Nas próximas seções são apresentadas as principais abordagens não intrusivas disponíveis na literatura e, ao final do capítulo, uma tabela de comparação qualitativa.

4.1 Project5

Uma das primeiras abordagens não intrusivas para detecção de anomalias em ambientes distribuídos foi o Project5 [1]. O objetivo principal do Project5 é prover um conjunto de ferramentas para depurar sistemas distribuídos em redes locais (LANs) sem que haja necessidade de se conhecer previamente as funcionalidades dos equipamentos instalados na rede. Mais especificamente, Project5 trata todos os equipamentos como caixas pretas e analisa apenas os dados que trafegam pela rede.

Considerando que a comunicação entre os componentes de rede pode ocorrer em vários níveis e que Project5 analisa a comunicação utilizando apenas os pacotes gerados pelas aplicações, foram propostos dois algoritmos de inferência: o *nesting*, que utiliza a temporização de pacotes RPC, e o *convolution*, que utiliza técnicas de processamento de sinais. O objetivo desses algoritmos é fornecer uma forma de análise direcionada que aponte os principais pontos de gargalo encontrados nas aplicações.

A extração dos padrões de comunicação utilizando somente informações dos pacotes de comunicação é uma tarefa difícil, principalmente pela quantidade de interações e de ruídos de rede (atrasos, retransmissões, etc) que atrapalham a correta inferência dos relacionamentos nas comunicações. Entretanto, uma observação importante é que quando problemas estão acontecendo, comportamentos diferentes dos comuns ocorrem com maior frequência. Dessa forma, é possível identificar os nós que participam desses padrões e que adicionam latências significativas.

As ferramentas e os algoritmos do Project5 mapeiam os pontos de latência e auxiliam os programadores a entender quais pontos estão aumentando o tempo de resposta de um sistema. Em resumo, essas ferramentas buscam:

- encontrar os padrões de caminho que causam alto impacto na comunicação;
- identificar quais nós participam desses padrões.

Para representar o caminho de comunicação, Project5 modela o ambiente de rede como um grafo em que cada nó é um equipamento e cada aresta representa uma ou mais conexões lógicas entre dois equipamentos. A Figura 4.1 ilustra uma requisição de um usuário que afeta diversos nós na rede (servidor web, autenticação, aplicação e banco de dados). Nesse caso, a ordem dos acessos é importante para relacionar o impacto causal em cada fase da requisição. Por exemplo: Cliente realiza requisição, servidor *web* consulta credenciais e assim por diante. Quando uma requisição externa é realizada, ela normalmente causa uma atividade no grafo ao longo do caminho causal.

O Project5 funciona em três fases: coleta dos traços de comunicação, inferência dos caminhos causais e dos padrões de comunicação e visualização dos dados. A primeira fase é *online* e captura todas as mensagens entre os nós. A segunda fase é *offline* e é a fase em que as informações são processadas (com algoritmos estatísticos de extração de relacionamentos) para inferir os caminhos de comunicação e o relacionamento das mensagens. A terceira fase é simplesmente a visualização dos resultados.

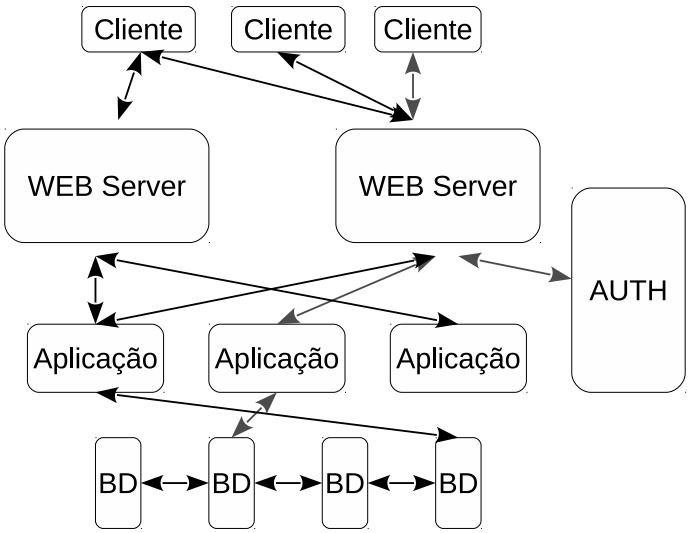


Figura 4.1: *Grafo mostrando um dos caminhos possíveis para uma requisição e os nós que foram acessados [1].*

Para realizar o mapeamento dos dados, são utilizadas tuplas para registrar informações específicas das mensagens. Cada tupla contém pelo menos três componentes: *timestamp*, *emissor* e *receptor*, mas pode incluir informações adicionais se houver necessidade, como, por exemplo, se é uma chamada ou retorno de RPC.

Como mencionado anteriormente, o Project5 utiliza dois algoritmos de inferência: o *nesting*, que trabalha com mensagens bem definidas do tipo RPC; e o *convolution*, que é um algoritmo mais genérico baseado na teoria de processamento de sinais. O primeiro examina as mensagens analisando suas relações em fluxos individuais. A Figura 4.2 exemplifica um caminho causal de comunicação inferido pelo algoritmo. Já o segundo algoritmo procura a relação causal entre várias mensagens agregadas à procura de padrões de comunicação.

Para encontrar o relacionamento entre as mensagens de comunicação, o algoritmo *nesting* combina todos os traços de rede em um único traço global. Depois ele examina cada um dos traços individuais para inferir quais chamadas estão aninhadas. Mais especificamente, se um nó A requisita um nó B que por sua vez requisita um nó C antes de B retornar para A, então a comunicação $B \rightarrow C$ está aninhada dentro da comunicação $A \rightarrow B$. A Figura 4.3 possui as chamadas da figura anterior ordenadas no tempo. Ao executar o algoritmo *nesting* neste grafo as chamadas $B \rightarrow C$ e $B \rightarrow D$ estão aninhadas na chamada $A \rightarrow B$.

O algoritmo *nesting* é dividido nas quatro fases listadas abaixo:

1. Encontrar todos os pares de chamadas no traço de rede;

2. Encontrar todos os possíveis aninhamentos de cada par de chamadas em outras e estimar a probabilidade de que cada candidato esteja ou não aninhado na chamada;

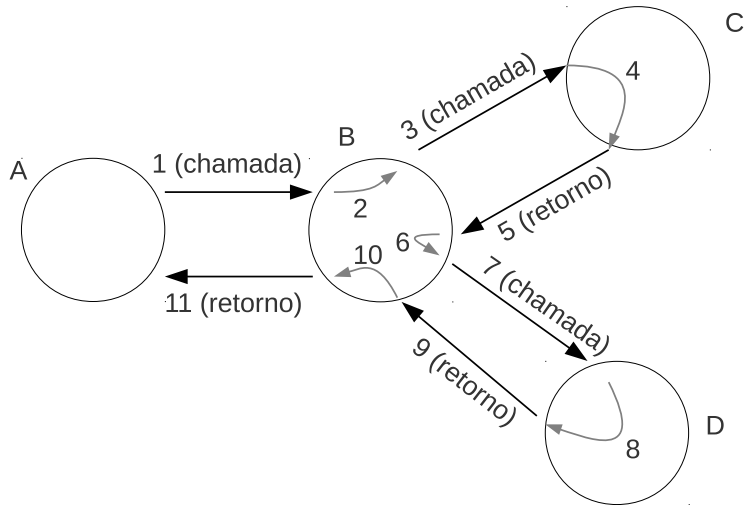


Figura 4.2: *Exemplo de um caminho causal de comunicação, os números representam a ordem das interações [1].*

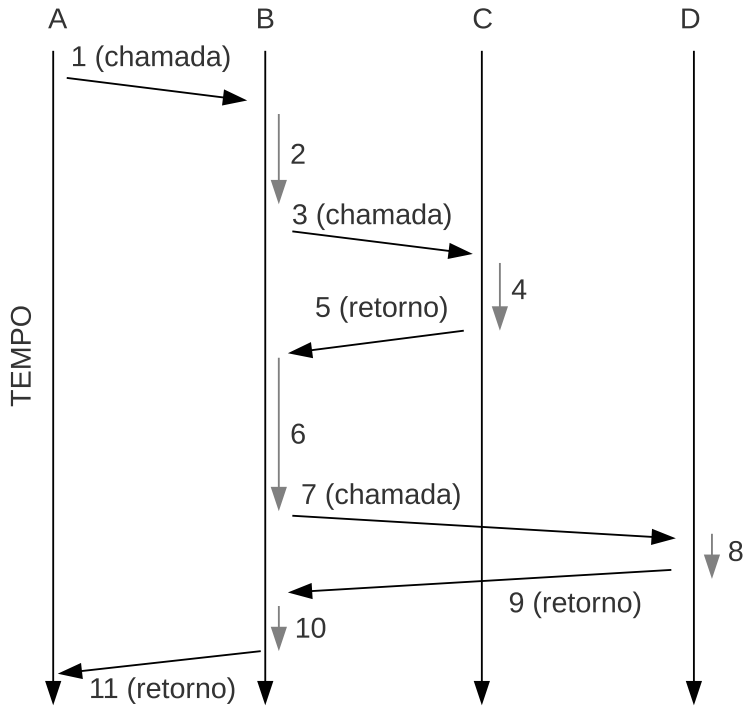


Figura 4.3: *Exemplo de caminho causal de comunicação em relação ao tempo de acontecimento das requisições [1].*

3. Selecionar o candidato com maior probabilidade de ter ocasionado a chamada para cada par de chamadas;

4. Derivar os caminhos de chamada dos relacionamentos de causa.

Além das quatro fases listadas, há alguns detalhes de implementação que precisam ser tratados nesse algoritmo, como, por exemplo, a identificação correta dos pares de chamadas. Problemas como perdas de pacote e retransmissões, em que mensagens podem ser duplicadas, interferem na identificação correta dos pares. O algoritmo *nesting* utiliza algumas heurísticas para minimizar essas e outras mensagens estranhas, mas infelizmente as heurísticas não conseguem tratar corretamente quando mensagens são descartadas durante as comunicações.

Já o algoritmo *convolution*, diferente do anterior, procura as relações de causa levando em consideração a agregação de múltiplas mensagens ao invés de as examinar individualmente. Ele separa todo o traço do sistema em um conjunto de traços por aresta em que são agrupados os eventos relativos às comunicações entre pares específicos de nós. Os traços agrupados por arestas são tratados como um sinal de tempo. A ideia central do algoritmo é converter os traços em sinais de tempo e usar técnicas de processamento de sinal, como cálculo de convolução, para encontrar o correlacionamento entre os sinais. Os resultados do *convolution* são grafos dirigidos de relacionamento em que cada nó pode aparecer várias vezes (exemplo: $A \rightarrow B \rightarrow A \rightarrow C$).

Embora os resultados apresentados pelo Project5 sejam significativos, principalmente por se tratar de um dos trabalhos pioneiros na área de análise de anomalias, há várias questões que não foram devidamente resolvidas. Um dos principais problemas dos algoritmos propostos é a forte dependência de relógios sincronizados nos equipamentos da rede. Os algoritmos dependem fortemente da utilização do NTP para realizar a agregação e evitar a duplicação de dados. Além disso, os nós devem possuir identificadores únicos em todos os traços coletados para se evitar duplicações e garantir que as informações sejam conciliadas corretamente.

4.2 Wap5

Assim como o Project5, o Wap5 [34] analisa os pacotes de rede e considera as aplicações como caixas pretas. O objetivo é extrair a estrutura causal de comunicação dentro das aplicações distribuídas e quantificar o atraso de processamento e de comunicação para identificar possíveis pontos de gargalo na rede.

A execução do Wap5 consiste em quatro passos, detalhadas a seguir. Primeiro é instalada uma biblioteca de interposição (*LibSockCap*) que coleta as informações das chamadas dos processos da aplicação por meio da API *socket*. Depois, os traços coletados são conciliados para formar um único traço com mensagens contendo os tempos (*timestamps*) do emissor e receptor quantificando e compensando a diferença de tempo e latência da rede. Em seguida é executado o algoritmo de análise de caminhos causais (*linking*) para con-

ciliar os traços e encontrar os prováveis caminhos que representam a causa do problema. Finalmente, são renderizados os caminhos causais como árvores ou linhas do tempo.

O Wap5 utiliza um modelo de causalidade muito simples. Uma mensagem A produz uma mensagem B, se A foi recebida pelo *host* X e B foi enviada pelo *host* X. Dessa forma é possível realizar duas análises: ou o pacote B foi influenciado pela chegada de A a X, ou ele foi gerado pelo próprio *host* X (geração espontânea) para um outro nó da rede. É importante ressaltar que o Wap5 não leva em consideração o modelo de causalidade em que vários pacotes de chegada geram um único de saída. Outro detalhe importante é que o pacote de saída B tende a ser sempre associado ao último pacote que chegou ao *host* X, mas nem sempre isso é verdade, resultando assim em inferências incorretas.

Os relacionamentos de inferência descritos anteriormente possuem comportamentos bem definidos e são normalmente síncronos. Quando acontecem eventos assíncronos, estes tendem a ser espontâneos e não têm relacionamento com eventos imediatamente anteriores. Uma das limitações do Wap5 é considerar temporizadores assíncronos como gatilhos de inferência.

A utilização da *LibSockCap* para captura de dados faz com que as mensagens sejam montadas da mesma forma que o programa as enxerga. Isso facilita a análise das chamadas de sistema e os relacionamentos que são gerados. Desta forma, é possível analisar o *host* e cada processo ou *thread* como um objeto diferente. Entretanto, as informações de controle dos pacotes perdem importância, pois a análise é feita em uma camada superior mais próxima da aplicação.

As desvantagens desta forma de captura são que alguns problemas de rede, como fragmentação e retransmissão, não são devidamente capturados. Nesses casos, a fragmentação na camada IP e a retransmissão na camada TCP não são visíveis para a biblioteca de interposição. Outro problema é quando um processo propositalmente adiciona um atraso após a captura do *timestamp*. Nesse caso, o atraso é atribuído à rede e não ao processo.

Uma vez que os dados foram coletados, o algoritmo de reconciliação converte o conjunto de atividades de *sockets* por processo (mensagens de rede e interprocessos) em uma única e mais abstrata mensagem entre nós internos. O algoritmo inclui a tradução dos eventos de *socket* para nomes de pontos-finais de fluxo (representando cada descritor de arquivo). A saída é um traço contendo tuplas de mensagens na forma (timestamp-emissor, ponto-final-emissor, nó-emissor, timestamp-receptor, ponto-final-receptor, nó-receptor).

Para determinar a latência de comunicação entre dois nós é utilizado o *timestamp* capturado do traço final da rede. Nesta etapa, é executada a conciliação que agrega caminhos com instâncias similares em padrões de caminhos. A forma mais simples é combinar os caminhos com estruturas idênticas de comunicação, por exemplo, servidores com clientes.

O Wap5 é capaz de determinar quando um nó da rede é um cliente ou um servidor. A lógica utilizada considera inicialmente que todos os nós são clientes e utilizam portas efêmeras. Durante a comunicação, se várias requisições de portas efêmeras apontam para uma única porta destino, esta porta se torna uma porta fixa e o nó é considerado um servidor.

Após a conclusão dessa análise preliminar, é executado o algoritmo *linking* para determinar se uma mensagem foi gerada espontaneamente ou como causa de outra mensagem. Como na maioria dos casos a causa é ambígua, o *linking* atribui para cada caso uma probabilidade. Dessa forma, são construídas instâncias de caminhos em que cada instância recebe uma pontuação que representa o produto de todas as probabilidades de conexão. Se uma conexão é ambígua com pontuação próxima de $0,5$ então dois caminhos são criados: um com o caminho ambíguo e outro sem (Figura 4.4).

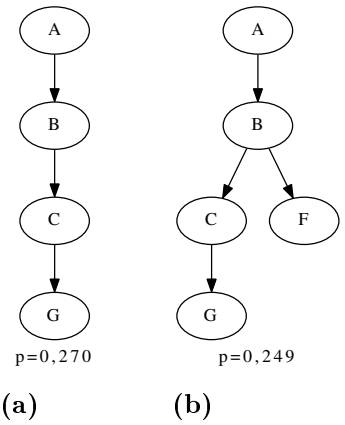


Figura 4.4: *Caminhos causais prováveis partindo do nó A e passando pelo nó B com suas probabilidades [34].*

Para determinar quando pacotes são gerados espontaneamente, é definido um intervalo de tempo x entre a chegada e saída de um pacote. Se a diferença de tempo superar este intervalo o pacote é espontâneo. Quando um pacote não é espontâneo, o problema é reduzido a vincular os pacotes de chegada e saída. Wap5 assume que o tempo de serviço (diferença de tempo entre a chegada e a saída) segue uma distribuição exponencial $f(t) = \lambda e^{-\lambda t}$, em que λ é o valor a ser encontrado. O valor de λ é definido com base na análise do traço de rede e corresponde à média dos tempos entre as chegadas de pacotes com a saída de pacotes mais próximas. A média não deve ultrapassar o valor estipulado de geração espontânea x.

O processo a seguir deve ser executado em todos os pacotes do traço. Para cada par de pacotes de saída $B \rightarrow C$ é estimado o atraso de serviço que B deve esperar (tempo de processamento do *host* X) para gerar C. Para a correta geração dos tempos de serviço é necessário saber quais são as mensagens geradoras, pois, desta forma, o *linking* calcula a média do atraso entre o pacote de chegada mais próximo e o pacote de saída desejado. Em cada mensagem $B \rightarrow C$ o *linking* procura em B a última mensagem encontrada e a adiciona à média do atraso. Ou seja $\lambda_{B \rightarrow C} = 1/d_{B \rightarrow C}$, em que $d_{B \rightarrow C}$ é a média estimada que o nó B espera antes de enviar a mensagem para C. É importante garantir que as mensagens geradas espontaneamente tenham o valor x (variável que armazena o tempo de geração espontânea) maior que o maior atraso real coletado nos traços.

Com o valor $\lambda_{B \rightarrow C}$ estimado para cada par de nós $B \rightarrow C$, o algoritmo atribui a cada pacote de chegada um peso baseado no seu atraso. Suponha que um pacote A tenha chegado ao nó X em t_1, B em t_2, C em t_3 e D em t_4 e o pacote K tenha saído do nó

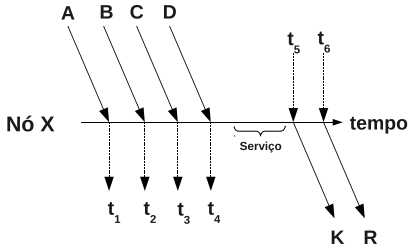


Figura 4.5: *Pacotes chegando e saindo do* host *X em tempos distintos [34].*

X no tempo t_5 como na Figura 4.5. Então para cada pacote proveniente de A, B e C, será atribuído um peso relacionando sua chegada à saída de X. O cálculo do peso usa a fórmula: $f(\Delta t) = e^{-\lambda_{X \to K}(\Delta t)}$ em que cada tempo t usa um intervalo de 1s e $\lambda = 1$.

$$A \to X : f(t_5 - t_1) = e^{-\lambda_{X \to K}(t_5 - t_1)} = 2,71^{-1*4} \approx 0,018$$

$$B \to X : f(t_5 - t_2) = e^{-\lambda_{X \to K}(t_5 - t_2)} = 2,71^{-1*3} \approx 0,049$$

$$C \to X : f(t_5 - t_3) = e^{-\lambda_{X \to K}(t_5 - t_3)} = 2,71^{-1*2} \approx 0,135$$

$$D \to X : f(t_5 - t_4) = e^{-\lambda_{X \to K}(t_5 - t_4)} = 2,71^{-1*1} \approx 0,368$$

$$\text{Espontâneo} : f(5) = e^{-\lambda_{X \to K}(5)} = 2,71^{-1*5} \approx 0,007$$

O último valor calculado $f(5)$ representa a probabilidade do pacote gerado ser espontâneo, como ilustrado na Figura 4.6. Com esses dados calculados, é necessário normalizar os pesos para representar as probabilidades de relação entre os pacotes de entrada com o de saída. No processo de normalização a soma de todas as probabilidades deve totalizar 1. Mais especificamente:

$$A \to X : 0,018/0,57 \approx 0,031$$

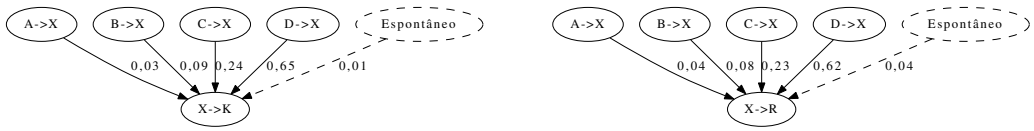
$$B \to X : 0,049/0,57 \approx 0,086$$

$$C \to X : 0,135/0,57 \approx 0,237$$

$$D \to X : 0,368/0,57 \approx 0,646$$

$$\text{Espontâneo} : 0,007/0,57 \approx 0,012$$

O último passo do *linking* é a construção dos caminhos utilizando os enlaces individuais e agregando-os em padrões. Na Figura 4.6, é possível relacionar os caminhos que foram criados na Figura 4.7 em que algumas possibilidades de caminhos foram geradas. O único caminho diferente foi o caminho $B \to F$, que possui probabilidade muito próxima a $0,5$ (50%) e por este motivo foi marcado como *tente-os-dois* para que sejam testadas as duas possibilidades de caminhos. O caminho $A \to B \to C \to G$ é o que possui as maiores probabilidades no primeiro trecho ($A \to B$ relacionado com $B \to C$), igual a $0,8$



(a) **(b)**

Figura 4.6: *Quatro chamadas em X que podem ter causado (a) $X \rightarrow K$ ou (b) $X \rightarrow R$ [34].*

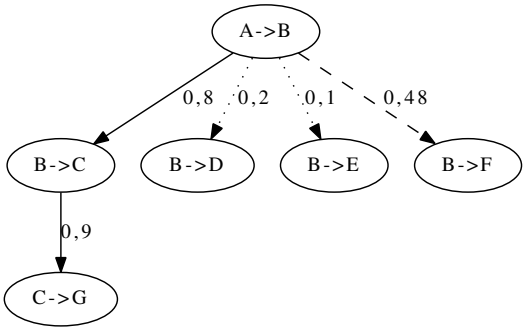


Figura 4.7: *Pesos normalizados como a probabilidade de um caminho ser o correto [34].*

e no segundo trecho ($B \rightarrow C$ relacionado com $C \rightarrow G$), igual a $0,9$. Este caminho está marcado com uma linha sólida e significa que é um caminho *provavelmente-verdadeiro*. Os caminhos pontilhados e probabilidades inferiores a $0,2$ são marcados como *provavelmente-falso* e não devem ser considerados. O único caso em que um caminho deve ser expandido é o exemplo da Figura 4.4, em que um caminho possuía probabilidade próxima a $0,5$ (*tente-os-dois*) e foi marcado como um caminho provável.

Para cada um dos caminhos é determinada a probabilidade de que este seja o correto. Esta probabilidade é calculada da seguinte forma: $p = x_1 * (1 - x_2)$, em que x_1 pertence a um caminho que contribui para a árvore gerada e $(1 - x_2)$ pertence a um caminho que não contribui, ou seja:

$$p_1 = 0,8 * 0,9 * (1 - 0,2) * (1 - 0,1) * (1 - 0,48) \approx 0,270$$

$$p_2 = 0,8 * 0,9 * (1 - 0,2) * (1 - 0,1) * 0,48 \approx 0,249$$

Quanto maior a árvore gerada, mais difícil é garantir que ela represente um caminho válido.

A visualização dos resultados gerados pelo *linking* são gráficos que representam o caminho completo da requisição, como aquele ilustrado na Figura 4.8. Os losangos representam os nós dos atrasos internos e externos que não puderam ser diferenciados, as caixas representam nós com o valor do seu atraso interno e as linhas, quando existirem, vão representar os canais de comunicação. O caminho causal é obtido percorrendo o gráfico da esquerda para a direita.

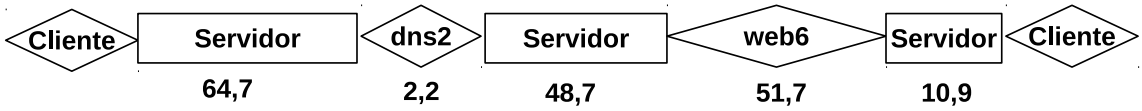


Figura 4.8: *Caminho causal da comunicação de um cliente a um servidor web através de um servidor de rede, que realiza uma consulta DNS [34].*

O Wap5 não foi projetado para funcionar em tempo real e sua análise depende fortemente dos *timestamps* dos pacotes. Esta limitação de vínculo com o tempo torna sua aplicação em ambientes de produção limitada, pois depende da sincronização de relógios. Além disso, o consumo de memória e a necessidade de realizar uma varredura inicial no traço para conciliação de informações é um empecilho na criação de uma versão com atualização dinâmica.

4.3 Sherlock

Uma abordagem recente e considerada uma das mais completas para detecção automática de anomalias em ambientes distribuídos foi proposta em [2]. Sherlock, como foi denominado o sistema proposto, fornece um conjunto de ferramentas para detectar a existência de falhas e problemas de desempenho por meio do monitoramento dos tempos de resposta dos serviços. Ele é capaz de descobrir os componentes envolvidos em uma requisição, incluindo elementos de balanceamento de carga e os mecanismos de *failover* mais comuns. Essas duas últimas funcionalidades são os pontos fortes do Sherlock e que o distingue dos demais sistemas propostos anteriormente.

Sherlock faz análise dos serviços de rede via inferência em múltiplos níveis de dependências e tem como objetivo detectar os problemas que frustram a experiência do usuário. Diferente de outras propostas, Sherlock não se preocupa em detectar eventos ou situações temporárias de sobrecarga que não interferem na percepção do usuário. Em algumas situações, um servidor pode estar apresentando alta utilização da CPU, mas os usuários não estão percebendo atrasos significativos nos serviços oferecidos. Esse tipo de anomalia não é considerada relevante para Sherlock.

Alguns modelos propostos na literatura [1, 34] tratam o estado de um serviço como uma tupla do tipo $(ativo, inativo)$, ou seja, o serviço está funcionando ou não. Entretanto, esse tipo de análise esconde informações como falhas ou problemas de desempenho importantes na percepção do usuário final. Sherlock melhora essa abstração modelando a disponibilidade de serviços em três estados diferentes: *ativo*, *problemático* e *inativo*. O primeiro significa que o serviço está ativo e respondendo dentro do esperado, o segundo significa que o serviço está enfrentando algum problema (não está respondendo, por exemplo) e o último que o serviço está fora do ar.

O mapeamento dos estados é feito com a utilização de um agente instalado em cada *host* da rede com a função de analisar os pacotes que são enviados e recebidos. Durante a análise dos pacotes, o agente também determina o conjunto de serviços dos quais o *host*

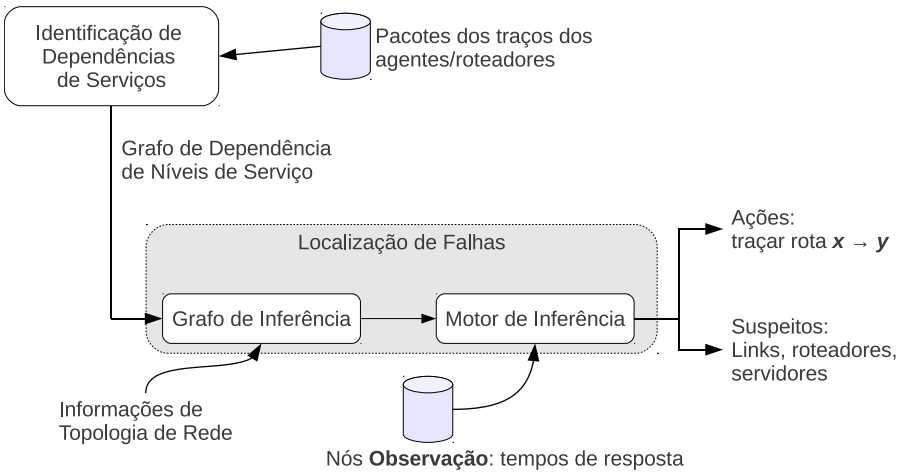


Figura 4.9: *Arquitetura de funcionamento de Sherlock [2].*

depende. Sherlock constrói automaticamente um Grafo de Inferência (GI) que captura as dependências entre todos os componentes da infraestrutura de TI combinando as visões individuais e completando as brechas que cada *host* tem da rede utilizando o conhecimento dos outros. A arquitetura geral do Sherlock pode ser vista na Figura 4.9.

As duas maiores contribuições de Sherlock são a descrição de um algoritmo de detecção de falhas chamado Ferret e a modelagem dos nós do GI com três estados. O algoritmo Ferret usa um modelo probabilístico para inferir falhas ou componentes que apresentam problemas utilizando observações de ambientes reais. O GI é rotulado, direcionado e fornece uma visão unificada das dependências dos *hosts*. A Figura 4.10 mostra um trecho do GI quando um usuário acessa uma página da internet.

Os nós deste grafo são de três tipos: os nós raízes de problema, que correspondem a componentes físicos que podem causar problemas na experiência do usuário, como servidores, roteadores ou *links* IP; nós de observação, que representam acessos a serviços de rede mensurados por Sherlock (estes nós modelam a experiência do usuário quando está utilizando um serviço); e finalmente os meta nós (*metanodes*), que interligam os nós raízes de problema e os nós de observação. Os meta nós se subdividem em três outros: Ruído-Máximo, Seletor e *Failover*. Os dois últimos meta nós são necessários para modelar balanceadores de carga (Seletor) e servidores que controlam redundância de serviço (*failover*).

O estado de cada nó no GI é expresso pela tupla (P_{ativo}, $P_{\text{problemático}}$, P_{inativo}). O estado P_{ativo} denota a probabilidade do nó estar funcionando, o estado P_{inativo} denota a probabilidade do nó estar experimentando uma falha/parada como, por exemplo, um servidor que parou ou um enlace rompido e o estado $P_{\text{problemático}}$ corresponde a probabilidade do nó estar com problemas como, por exemplo, um servidor sobrecarregado ou um enlace com altas taxas de erro. Neste último caso, o serviço está ativo mas os usuários estão experimentando baixo desempenho. A soma dos valores de $P_{\text{ativo}} + P_{\text{problemático}} + P_{\text{inativo}}$ deve ser igual a 1,0. Os nós raízes de problema são independentes de qualquer outro nó no GI e os nós de observação têm seus estados determinados pelos estados dos seus

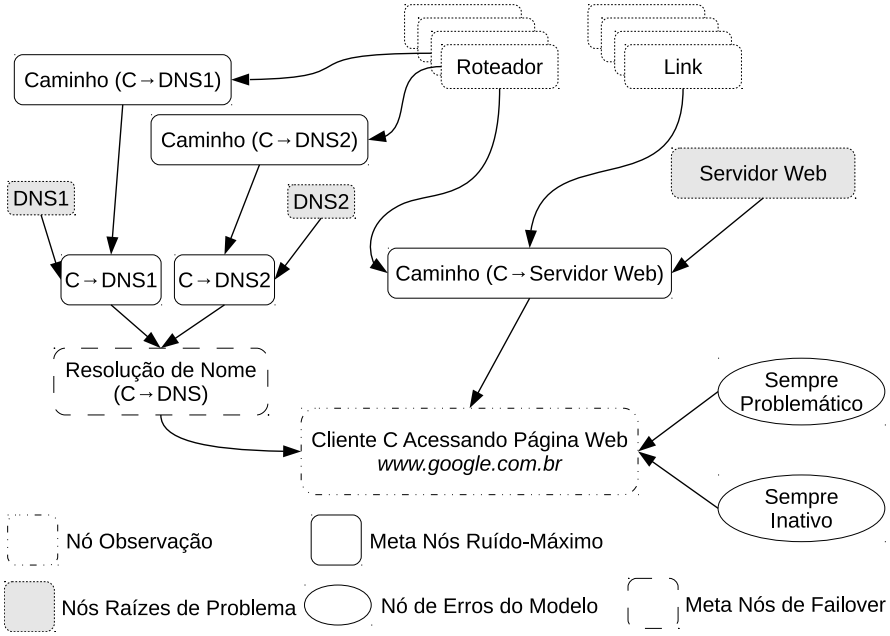


Figura 4.10: *Parte do GI gerado pelo Sherlock [2].*

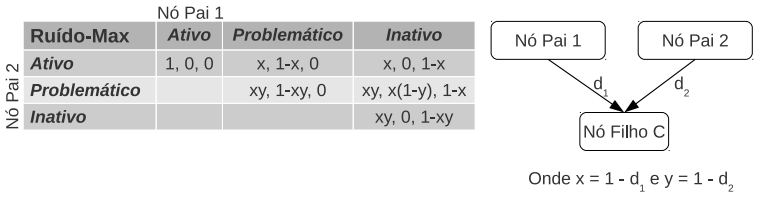


Figura 4.11: *Tabela verdade do modelo de meta nós Ruído-Máximo quando um filho tem dois pais. O inferior da tabela foi omitido por seguir o mesmo raciocínio da parte superior [2].*

antecessores no grafo. Esta compreensão da forma como os nós do GI se comportam é importante para entender como a propagação de estado entre nós pais e filhos funciona.

Uma aresta entre os nós A e B no GI codifica o grau de dependência entre o nó A e o nó B. Dessa forma, se o nó B está ativo significa que o nó A também está ativo. Para capturar as variações de força entre as dependências, as arestas do GI são rotuladas com a probabilidade da dependência, ou seja, uma probabilidade maior indica uma dependência forte e uma menor uma fraca. Existem também dois estados especiais que são vinculados aos nós raízes de problema: o estado sempre com problema (AT -*Always Troubled*) e o estado sempre inativo (AD - *Always Down*), utilizados para modelar fatores externos não identificados pelo modelo. O estado AT é definido pela tupla $(0, 1, 0)$ e o estado AD é definido como $(0, 0, 1)$.

Com o formato de dependência definido, é necessário propagar os estados dos nós raízes de problema para os nós de observação utilizando os meta nós, pois é crucial em um modelo probabilístico saber como os nós pais governam os estados dos nós filhos. Em

Nó Pai 1

Nó Pai 2 / Seletor	*Ativo*	*Problemático*	*Inativo*
Ativo	1, 0, 0	x, 1-x, 0	x, 0, 1-x
Problemático		0, 1, 0	0, x, 1-x
Inativo			0, 0, 1

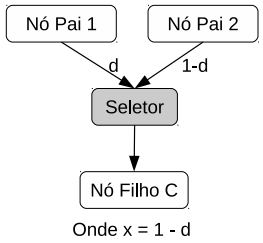


Figura 4.12: *Tabela verdade do modelo de meta nós Seletor quando um filho tem dois pais e o nó filho seleciona o Nó Pai 1 com probabilidade d e o Nó Pai 2 com probabilidade* $1 - d$*. A parte inferior da tabela foi omitida por seguir o mesmo raciocínio da parte superior [2].*

um exemplo, se os nós pais A e B tem probabilidade A(0,6; 0,4; 0) e B(0,2; 0,4; 0,6) é importante saber qual será o estado do filho destes nós no GI e a natureza da dependência (definida pelos meta nós).

Como a natureza das dependências depende dos meta nós, é importante entender como cada um deles funciona. O meta nó Ruído-Máximo é o mais comum de todos. O *máximo* quer dizer que se os pais de um nó estão inativos então o filho também estará. Se nenhum nó pai estiver inativo e qualquer pai estiver com problema, então o filho estará com problema. Se todos os pais estiverem ativos então os filhos também estarão. O *ruído* implica que se um nó filho não tiver probabilidade de dependência 1,0 com o nó pai significa que o filho tem a probabilidade de estar ativo mesmo que o pai esteja inativo (matematicamente esta probabilidade é representada por $1 - d$ e representa a chance que o filho tem de não ser afetado pelo estado do seu nó pai, ideia semelhante àquela utilizada no Wap5 [34] na seleção dos caminhos). A Figura 4.11 apresenta a tabela verdade para o meta nó Ruído-Máximo quando um nó filho tem dois nós pais.

O meta nó Seletor é utilizado para modelar cenários com balanceadores de carga. Modelar balanceamento com o Ruído-Máximo traria problemas, pois uma requisição de um cliente distribuída entre dois servidores, por exemplo, atribuiria uma probabilidade de dependência 0,5 para cada servidor (desde que cada servidor receba metade das conexões). Utilizando somente o meta nó Ruído-Máximo um cliente teria 25% de chance de estar ativo mesmo quando ambos os servidores estivessem inativos, o que é uma análise incorreta. A tabela verdade para os meta nós do tipo Seletor pode ser visualizada na Figura 4.12 e expressa o fato de que o filho está selecionando um dos servidores com probabilidade de 50%.

O último modelo de meta nó é o que captura um mecanismo comum em servidores usados em grandes empresas, que é o comportamento de *failover* (redundância). O *failover* é uma técnica de redundância que faz os clientes acessarem o servidor primário, quando este está ativo, ou o secundário (*backup*), quando o primário falha, evitando a indisponibilidade. Um exemplo de servidor que utiliza esta técnica é o DNS. O *failover* não pode ser modelado nem pelo Ruído-Máximo nem pelo Seletor, pois a probabilidade de acessar o servidor secundário depende exclusivamente da falha do servidor primário. Na Figura 4.13 está a tabela verdade para o modelo de meta nós *failover*. Enquanto o servidor primário

Nó Pai 1

Nó Pai 2 \ Failover	*Ativo*	*Problemático*	*Inativo*
Ativo	1, 0, 0	0, 1, 0	1, 0, 0
Problemático	1, 0, 0	0, 1, 0	0, 1, 0
Inativo	1, 0, 0	0, 1, 0	0, 0, 1

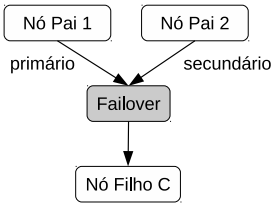


Figura 4.13: *Tabela verdade do modelo de meta nós* Failover *[2].*

estiver ativo ou problemático, o filho não será afetado pelo servidor secundário. Quando o servidor primário estiver no estado inativo, o nó filho continuará ativo desde que o servidor secundário esteja ativo.

As tabelas verdades são utilizadas pelo algoritmo Ferret na construção dos *vetores de atribuições*. O vetor deve especificar, por exemplo, que o *link*1 está problemático, o servidor2 está inativo e todos os outros nós raízes de problema estão ativos. O problema de localizar uma falha se torna equivalente a encontrar um vetor de atribuições que melhor explique as observações detectadas pelos usuários. O algoritmo Ferret tem como entrada o GI e as medidas associadas aos nós de observação (por exemplo os tempos de resposta). A saída do Ferret é uma lista ordenada pelo grau de confiança com a melhor representação que explique as observações dos usuários.

Se fossem analisados todos os r nós raízes de problema, o algoritmo teria um tempo exponencial de ordem $O(3^r)$. Buscando reduzir este tempo, foram feitos cortes no espaço de busca. O primeiro corte foi baseado no fato de que somente alguns nós raízes de problema estão com problemas ou inativos ao longo do tempo. O segundo corte avaliou que um nó raiz de problema é definido como ativo em um ou mais vetores de atribuição e desta forma a avaliação deste vetor só precisa ser reavaliada nos estados descendentes dos nós raízes de problema que estão inativos. Assim o algoritmo Ferret preprocessa o GI propagando a todos filhos de nós ativos o estado ativo até alcançar os nós de observação.

É importante entender a construção do GI e como ele é utilizado para localizar problemas. Sherlock consiste de um Motor de Inferências centralizado e agentes distribuídos na rede. Não é necessário realizar nenhuma modificação nos roteadores, aplicações ou *middlewares* (tornando esta solução não intrusiva). A Figura 4.9 ilustra o processo de três etapas de Sherlock.

Na primeira etapa é construído o Grafo de Dependências de Serviço (GDS). Em cada cliente os agentes de Sherlock são responsáveis por monitorar os pacotes de um ou mais *hosts* e mapear seus serviços e dependências. Os pacotes podem ser coletados do próprio *host* ou obtidos pela coleta de pacotes de enlaces ou roteadores próximos.

Com esses traços, os agentes computam as dependências entre os serviços e a distribuição dos tempos de resposta. Essas informações são repassadas ao motor de inferência que agrega as dependências entre os serviços para construir o GDS. Embora o GDS seja relativamente estável e mude somente quando novos *hosts* ou aplicações são adicionados à rede, o motor de inferência combina o GDS com a topologia de rede (utilizando o comando *traceroute*) para construir o GI unificado de todos os serviços. Por último, o mo-

tor de inferência executa o algoritmo Ferret sobre os tempos de resposta observados pelos agentes e o GI, buscando identificar os nós (raízes de problema) responsáveis por qualquer problema observado. Esses passos são executados toda vez que um agente detecta altos tempos de resposta em alguma aplicação de interesse.

Sherlock é uma ferramenta capaz de mapear os problemas de rede com base na visão do usuário. Ele realiza a detecção de problemas que prejudicam a experiência dos clientes de forma não intrusiva e utiliza somente os pacotes coletados na rede. Seus principais pontos fracos são a utilização do *traceroute* para construir a topologia da rede, o motor de inferência que é concentrado em um único ponto da rede reduzindo a escalabilidade de uso, a necessidade de auxílio de operadores externos para detectar balanceamento e *failover* e a explosão combinatória necessária na geração dos vetores de atribuição.

4.4 eXpose

O eXpose [22] é uma ferramenta de análise *offline* cujo objetivo é extrair as regras de comunicação e dependências utilizando o tráfego de rede. Ele utiliza um esquema de análise de pacotes não orientado (sem auxílio externo ou informações adicionais) e extrai regras das interações encontradas nos traços de rede ao contrário de ferramentas como NetFlow [9] e MRTG [32] que apenas contabilizam quantidades de pacotes. A ideia chave do eXpose é que um grupo de fluxos que constantemente aparecem juntos podem ser correlacionados. O desafio neste caso é aplicar esta ideia em conjuntos de milhões de fluxos de rede. Para resolver este problema o eXpose seleciona as regras que potencialmente possuem informações relevantes, remove os traços que não são importantes e utiliza medidas estatísticas apropriadas para pontuar as regras candidatas. Depois disso as regras descobertas são agregadas em um pequeno número de conjuntos que devem ser validados pelo administrador de rede. Embora o eXpose faça a extração das dependências sem guias, seus resultados precisam do auxílio humano para validar as extrações.

O eXpose possui algumas contribuições importantes como:

- Descoberta das regras de configurações e protocolos em uma rede LAN sem prévio conhecimento das aplicações;
- Descoberta das regras para a maioria das aplicações que geram tráfego na rede como: *email, web*, servidores de arquivo, aplicações P2P;
- Detecção de configuração e malwares na rede;
- Mineração de regras dos fluxos com um tempo menor que a duração dos traços de rede (o que torna possível o desenvolvimento futuro de uma versão *online*).

De acordo com o eXpose, as comunicações que ocorrem dentro dos traços de rede possuem mais informações quando são analisadas como conjuntos. Uma regra de comunicação $A \rightarrow B$ ocorre se, no fluxo do traço de rede, uma comunicação A implica em

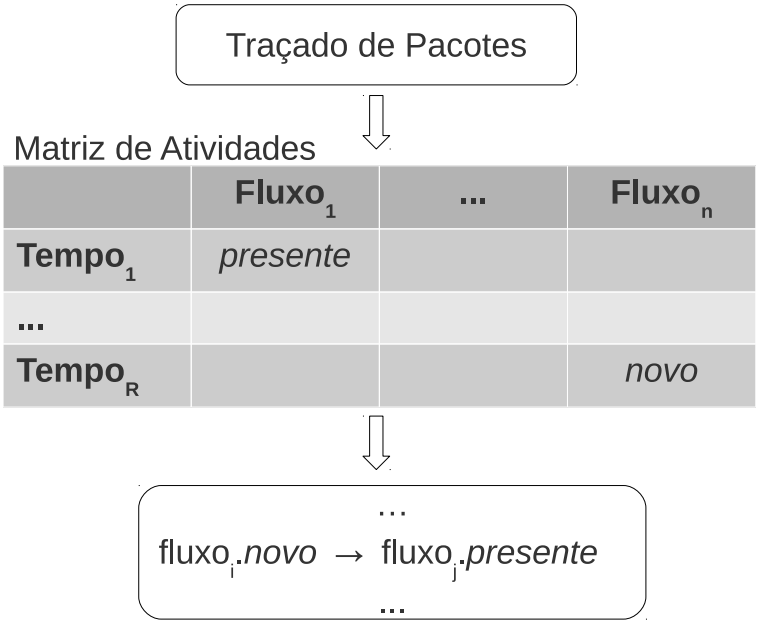


Figura 4.14: *Esquema geral do fluxo de trabalho do eXpose [22].*

uma comunicação B. Um exemplo prático é a requisição de uma página de internet pelo usuário, que antes de acessar a página (A) necessita da resolução de nomes via DNS (B).

Esses relacionamentos são difíceis de detectar sem o prévio conhecimento da aplicação. A extração destes relacionamentos baseia-se na identificação dos grupos de fluxos que ocorrem com alta frequência e se repetem constantemente. Como o eXpose foca na repetição de fluxos, quando as comunicações ocorrem com baixa frequência, o problema é resolvido utilizando regras genéricas com coringas para agregar mais fluxos e assim obter amostras mais representativas.

Para a extração das regras de comunicação, os traços de rede devem ser particionados em janelas de tempo (na execução do eXpose a janela foi de 1s), pois é dentro dessas janelas que são observadas as dependências. Com as janelas e os fluxos de tempo é construída a *Matriz de Atividade*, em que as linhas são representadas pelas janelas e as colunas pelos fluxos. Cada entrada (linha x coluna) na matriz corresponde aos valores: novo, presente ou ausente, denotando a atividade do fluxo (coluna) na janela de tempo (linha). A Figura 4.14 mostra o esquema geral de funcionamento do eXpose.

A construção das regras de comunicação é simples e segue o seguinte raciocínio: uma requisição $A \rightarrow B$ só existe se A e B ocorrerem juntos, ou seja, se $Prob(A \wedge B)$ é alta. Esta é a ideia mais simples para a construção de relacionamentos, porém este raciocínio nem sempre é correto e leva a muitos falsos positivos e negativos. A regra é que dois fluxos dependentes sempre ocorram juntos no traço. Um exemplo para o problema sugerido, é que fluxos podem ter alta probabilidade de ser dependentes simplesmente porque um deles é muito popular (como o *download* de um arquivo grande que aparece em várias janelas de tempo).

Desta forma, o eXpose constrói os conjuntos de regras candidatas do tipo $A \rightarrow B$ (em

que ambos A e B são tuplas <fluxo, atividade-do-fluxo>) analisando todo o traço. Depois de catalogadas, as regras serão pontuadas pelo resultado da execução do JMeasure [36] (algoritmo de mineração de dados) e caso A e B sejam independentes sua pontuação de relacionamento será zero.

Infelizmente, o JMeasure sem modificações é bom para mineração de dados genéricos mas ruim para identificar dependências de rede. Há pelo menos três problemas com o seu uso. O primeiro problema são as correlações negativas pois o JMeasure pontua relacionamentos equivocados com valores altos. A solução neste caso é utilizar somente as correlações positivas.

O segundo problema é relacionado aos fluxos longos (como o download de um arquivo grande da Internet), pois eles são representados por muitas entradas do tipo *presente* nas janelas de tempo. Este é o problema em que vários fluxos curtos acabam associados a um fluxo popular. Para evitar este problema não são reportadas regras envolvendo fluxos que acontecem no traço em mais de 90% das janelas de tempo.

O terceiro problema é com a quantidade de possibilidades (explosão de relacionamentos) na geração de regras de extração. A redução da quantidade de regras é simples: só são reportadas as regras em que os fluxos tenham pelo menos um endereço IP em comum. Além disso, a maioria dos fluxos acontece em apenas algumas janelas e com poucos dados. Por isso é necessário K fluxos ativos no traço de pacotes (K igual a 5000) para uma boa extração.

As dependências são sempre observadas aos pares. O problema é que nem todas as dependências ficam visíveis, por exemplo, quando a quantidade de amostras é pequena. O eXpose soluciona este problema criando as regras coringas já citadas. Para cada regra criada é também criada uma genérica. Por exemplo, na conexão de um cliente ao servidor *web* é criada a regra <Cliente.Porta : ServidorHTTP.80> representando esta conexão e associada a ela está a <*.* : ServidorHTTP.80>, que é uma regra genérica que aceita qualquer conexão com este servidor na porta 80.

As regras genéricas acumulam um número maior de amostras e com elas é possível ajustar melhor o relacionamento dos fluxos. No exemplo acima, a porta 80 é a padrão para o serviço *web*. O eXpose não fica limitado às portas conhecidas; ele cria uma lista de portas que são utilizadas por muitos fluxos e as define como portas de serviço (ideia semelhante ao Wap5 [34]). Um exemplo consiste em clientes P2P que são tanto clientes quanto servidores. Para as regras coringa é utilizada a quantidade de incidências que casam com a regra, se esta for maior que um gatilho β de três ocorrências, ela passa a ser reportada.

A complexidade do algoritmo para minerar e computar as regras considerando W janelas de tempo consecutivas e K fluxos ativos precisa considerar os $O(K^2)$ pares de fluxo. Isso levaria tempo $O(W * K^2)$ e o termo quadrático dominaria o tempo de execução. O eXpose observa que em qualquer janela de tempo só existem K fluxos ativos e, por isso, ao invés de contabilizar a frequência de todos os K^2 pares de fluxos da janela de tempo, são contabilizados somente os pares de fluxos ativos. Se a $w^{\text{ésima}}$ janela tem S_w fluxos o eXpose computa as frequências de todas as regras com tempo $O(\sum_{w=0}^{W} S_w^2)$.

Para observar as regras envolvendo N fluxos é necessário tempo $O(W * K^N)$. Isso acontece porque é preciso verificar cada um dos K^N grupos de fluxo gerados em cada uma das W janelas do traço. Espera-se que regras complexas tenham baixa ocorrência (esperança de *Ocams razor*) e, por isso, a complexidade computacional deve se manter estável com o aumento dos fluxos.

Embora o eXpose seja *offline*, como seu tempo de execução foi inferior ao tempo de coleta talvez seja possível a construção de uma versão *online*. Seria apenas necessário manter contadores de estado e frequência para identificar os K fluxos mais ativos.

A abordagem do eXpose busca uma forma diferente de agregar as informações coletadas da rede e atribuir um significado sem conhecimento prévio das aplicações. Enquanto no Wap5 [34] espera-se que o tempo de processamento dentro dos *hosts* siga uma distribuição exponencial, no eXpose busca-se alinhar o conhecimento da rede aos grupos de fluxos que se relacionam.

A garantia destes relacionamentos utiliza a medida de mineração de dados JMeasure modificada para pontuar os relacionamentos interessantes. Esta medida mostrou-se mais eficaz que a distribuição exponencial pois, no geral, o tempo de processamento, salvo nas situações de ruído na rede, tende a ser estável (desde que não envolva interação com o usuário).

4.5 Constellation

Constellation [3] é uma ferramenta para automatizar a descoberta dos relacionamentos da rede, serviços e construir um mapa entre estes relacionamentos. É uma ferramenta que utiliza informações compartilhadas no processo de detecção de anomalias. O mapeamento é feito utilizando técnicas estatísticas em cada par sem informações prévias. Como resultado das construções, qualquer computador pode obter uma visão global de como os serviços estão relacionados.

A maneira mais fácil de entender o funcionamento do Constellation é observar o acesso de um cliente a um serviço de internet. Utilizando uma estação de trabalho qualquer, é construída uma constelação que representa o acesso desta estação à internet utilizando um *proxy*. As relações da estação com o *proxy* de internet e deste para os *proxys* de rede são mapeadas como mostra a Figura 4.15. Quando um dos *proxys* para de responder (dados não fluem mais nos dois sentidos) o cliente experimenta lentidão em alguns acessos (Figura 4.16). Esta lentidão é comprovada pela modificação da constelação, em que um dos fluxos está apenas na direção de saída.

Outro exemplo interessante é a reconfiguração da topologia de servidores. Um servidor de banco de dados pode ser acessado por vários serviços e sistemas e o administrador pode desconhecer o impacto causado pela troca do endereço IP do servidor. Com a constelação de acessos ao banco, é possível avisar com antecedência todos os clientes sobre a modificação, reduzindo a quantidade de problemas causados. Dessa forma, Constellation simplifica a busca por dependências em todas as direções (de quem o par depende e para quem depende do par).

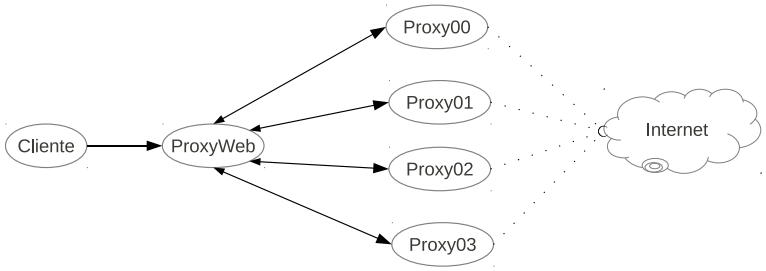


Figura 4.15: *Constelação da primeira hora das requisições HTTP encaminhadas para* 4 *servidores* proxy *[3].*

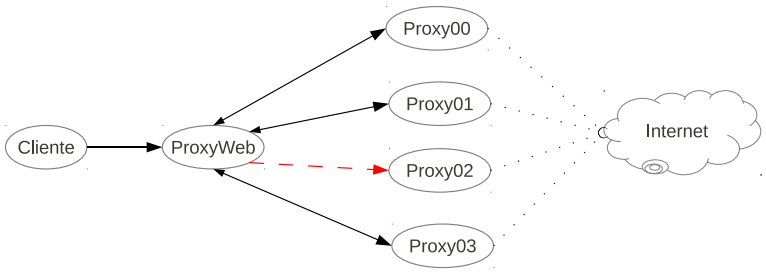


Figura 4.16: *Constelação da segunda hora das requisições HTTP encaminhadas para* 4 *servidores* proxy *com falha em um dos servidores [3].*

A constelação é construída utilizando "blocos de montagem". Cada par da rede participa na construção da constelação executando um serviço que aprende todos os relacionamentos a partir do tráfego de rede originado ou finalizado no par. Além disso, cada par permite que seus vizinhos realizem consultas em seus dados e, desta forma, é possível explorar todas as dependências da rede apenas varrendo os pares.

A construção da constelação implica na execução de alguns passos, como ilustrado na Figura 4.17. Primeiro, os pacotes da rede são agrupados em *canais* unidirecionais identificados pela direção (saída ou chegada no par), por uma regra, pelo serviço (protocolo, aplicação ou serviço de rede), pelo par remoto e a porta do cliente. Depois esses canais são *agregados* permitindo o tratamento do tráfego de vários pares como origem ou destino em uma única entidade. Logo em seguida são aplicadas relações de tempo para detectar os relacionamentos esperados entre as mensagens, removendo os ruídos e rajadas e mantendo somente o sinal. Com os canais de entrada e saída de tráfego determinados, o Constellation utiliza um conjunto de modelos probabilísticos que melhor descrevam os relacionamentos de tempo observados.

Para encontrar a melhor modelagem, é utilizado o algoritmo *Maximização de Expectativa* (EM) [5] que ajusta os modelos probabilísticos registrados com as informações coletadas na rede.

Constellation utiliza a mesma ideia base do Wap5 [34], em que é possível extrair informações relacionando os pacotes de entrada e saída de um par da rede. Como Constellation também correlaciona pacotes em mais de um nível, é necessária uma condição de parada.

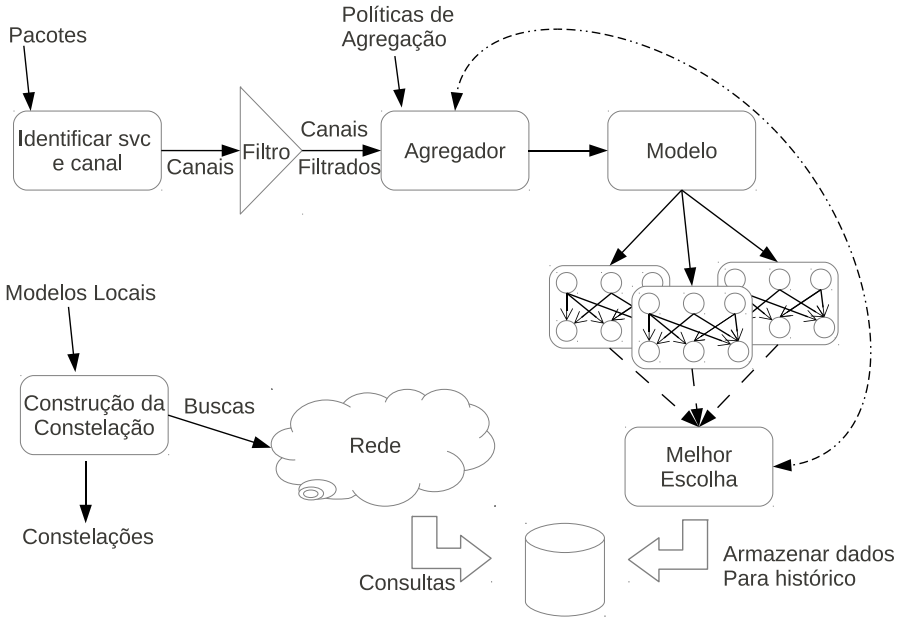


Figura 4.17: *Estágios de processamento do serviço local do Constellation. Pacotes de um mesmo serviço e pares são agrupados em canais, que são opcionalmente filtrados e agregados. Os relacionamentos entre entradas e saídas dos canais são "encaixados" no melhor modelo probabilístico. Cada par exporta seus dados para um banco de dados que permite consultas externas de qualquer par para a construção da constelação [3].*

Por isso, a cada salto entre os pares é atribuída uma pontuação que leva em consideração a distância em relação ao par origem da constelação. A busca termina quando não existem mais canais de saída relacionados ao canal de entrada observado.

No Wap5, os pacotes de entrada e saída são relacionados por uma distribuição exponencial. No Constellation, o modelo probabilístico utilizado é o CT-NOR (*Continuous Time NoisyOr*) que analisa cada canal de saída por vez e determina o canal de entrada que melhor o explica. Não existe diferença na análise de pacotes para o Wap5 ou o Project5, pois no Constellation é esperado que a relação de tempo entre os pacotes k, no canal de entrada j, que gera o pacote l de saída, em uma janela fixa, esteja de acordo com a distribuição de Poisson. Entretanto, Constellation não infere somente um único caminho causal como nos dois trabalhos citados. Ele é capaz de identificar todos os pares de canais relacionados em cada par, permitindo uma visão completa das dependências de todos os serviços.

4.6 Netmedic

Netmedic [24] foi modelado a partir da análise de redes de pequenas empresas utilizando os resultados extraídos dos principais problemas detectados pelas equipes de suporte. As principais técnicas para mapear os problemas no Netmedic são: organizar os problemas de forma detalhada como um GI e estimar quando duas entidades de rede estão causando

impacto entre si.

O GI é granular o suficiente para mapear componentes como processos ou configurações de aplicações. Diferente de outras abordagens, o estado de um componente de rede é composto por várias variáveis para avaliar a saúde (estado considerado normal) de uma aplicação. Essas variáveis podem, por exemplo, incluir informações de utilização de recursos, tempo de resposta a requisições e outros dados específicos da aplicação (como a quantidade de respostas que são códigos de erro).

Netmedic recebe como entrada as dependências de todos os processos ativos de cada máquina em um *template* e automaticamente constrói o grafo de dependências de cada componente. Ele implementa uma lógica temporal baseada nos eventos passados que é considerada robusta para análises de dados realistas. Com o conhecimento temporal dos eventos são feitas podas no GI utilizando detecções de anormalidades nos relacionamentos das aplicações. Estas detecções utilizam cálculos estatísticos para facilitar o mapeamento dos pontos de mudança que afetam componentes ou os componentes que não causam mudanças (retirados do grafo por outra poda).

A precisão do Netmedic na detecção dos problemas é atribuída ao mapeamento lógico da coleta de informações dos chamados requisitados à equipe de suporte da organização. Dos chamados são extraídos grupos lógicos de acordo com os sintomas observados e as causas relacionadas aos problemas. Dessa forma, é possível construir o grafo da rede modelando as dependências entre processos das aplicações, elementos de rede e arquivos de configuração. Na representação do grafo, dois componentes estão conectados por uma aresta direcionada se um causa impacto no outro.

Uma vez que o estado dos componentes da rede é visível, se algum mudar será possível identificar quais são os prováveis responsáveis pela mudança. Cada causa tem as seguintes propriedades: mudanças de visibilidade que podem explicar efeitos observados e mudanças de visibilidade que não podem explicar as mudanças de visibilidade em outros componentes. Na Figura 4.18 o computador saudável está sendo afetado pelo problemático ao tentar acessar um serviço remoto. Embora o problema seja causado pelo computador problemático (que está sobrecarregando o servidor), na visão do computador saudável, o problema é o servidor que está com alto tempo de resposta para as suas requisições.

A dificuldade na estimativa dos componentes que causam impacto é que as interações entre os vários componentes são desconhecidas. A solução foi a criação de uma primitiva que utiliza o histórico dos acontecimentos. Esta primitiva usa as informações do histórico de junções dos componentes para estimar o quanto um componente causa impacto no seu vizinho (este valor é atribuído à aresta como o peso do impacto de um componente no outro no grafo de dependências). Esses pesos são utilizados para identificar as prováveis causas que apontem para o componente afetado. Para entender como funciona esta primitiva, é necessário entender como duas variáveis se comportam ao longo do tempo. Por isso, é importante saber como os componentes A_1 e B_1 estão se comportando ao longo do tempo e se o comportamento verificado é recorrente. De forma empírica, se os comportamentos forem similares ao comportamento atual A_{atual} e B_{atual}, um grau de confiança e um peso são atribuídos a este relacionamento. Da mesma forma, se os comportamentos não forem similares então as chances de A_1 e B_1 estarem relacionados reduz significativamente com

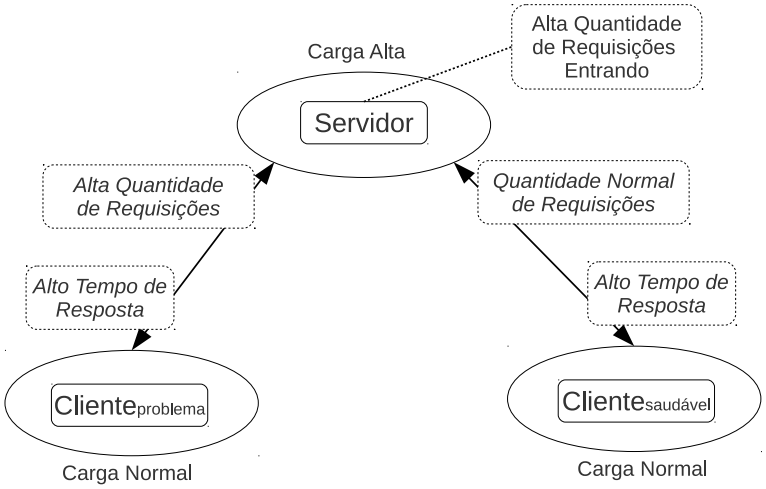


Figura 4.18: *Ilustração de um problema em que a visibilidade não afeta diretamente a mudança de visibilidade de outros componentes [24].*

o tempo. Isso significa que, se existir uma perturbação em A_1 e não existir qualquer mudança atual ou histórica em B_1, a probabilidade de A_1 e B_1 estarem relacionados é muito pequena.

Netmedic recebe algumas informações adicionais além do *template* que mapeia as aplicações como tipos de componentes e fontes de dados. Cada uma dessas informações de entrada é utilizada para criar um conjunto que será utilizado na organização de uma lista de possíveis causas para o atual estado problemático de um componente. A saída do Netmedic é uma lista com os possíveis causadores do problema para cada um dos componentes afetados. O funcionamento geral do Netmedic pode ser visualizado na Figura 4.19.

Existem três peças fundamentais no funcionamento do Netmedic: a captura do estado dos componentes de rede, a geração do grafo de dependências baseada no diagnóstico do estado dos componentes e o grafo de dependências. Existem várias formas de particionar a rede em componentes e no Netmedic foram utilizados os *logs* de falhas gerados pelos usuários. O Netmedic também considera os processos das aplicações, as máquinas e os *firewalls* em execução como componentes prováveis para uma falha.

Para garantir que mesmo após várias reinicializações do sistema exista uma forma concisa de mapeamento de processos, Netmedic utiliza o caminho completo da linha de comando de uma aplicação ao invés do identificador de processo no sistema operacional. Com o processo mapeado, Netmedic armazena as informações do estado de cada um deles por meio da união de algumas informações como: recursos consumidos e tráfego de rede (dados mais genéricos) e frações de falhas de requisições ou quantidade de requisições de um tipo específico (variáveis específicas).

Após a coleta das informações, são feitas manipulações para construção do grafo de dependências. Para modelar a rede como um grafo de dependências é necessário criar arestas que conectem os componentes quando um for diretamente dependente do outro. O *template* de entrada do Netmedic tem um componente central cercado por outros com-

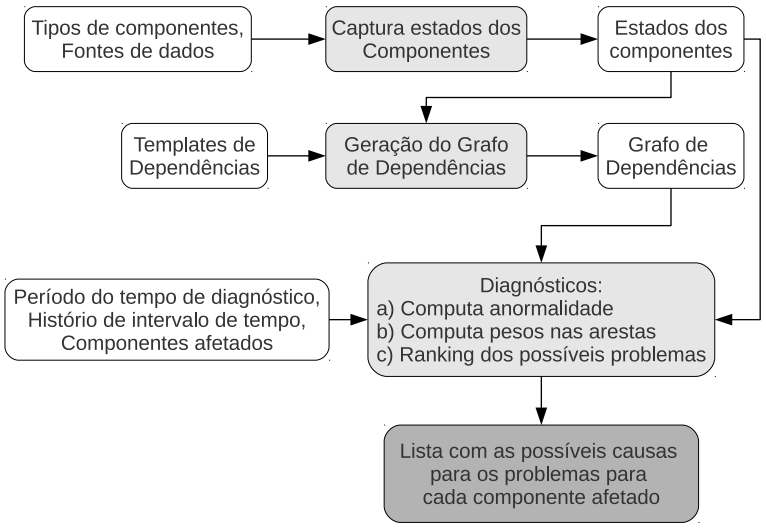


Figura 4.19: *Fluxo de funcionamento geral do Netmedic [24].*

ponentes e estes componentes ao redor causam impactos no central. Na prática, as arestas do grafo de dependências correspondem às arestas dos *templates*.

Netmedic analisa somente as comunicações que estejam envolvidas em troca de pacotes IP (ou seja, não funciona com trocas por memória compartilhada). A comunicação IP é capturada utilizando o conjunto de vizinhos de cada nó e depende das configurações do *firewall* local e remoto. No mapeamento do caminho da rede entre duas máquinas existe a dependência de todas as máquinas que injetam tráfego neste caminho e todos os outros tráfegos monitorados fora da rede analisada. Os únicos que não têm relação de dependência direta são os *templates* que representam componentes de configuração. Se uma configuração é modificada e representa a melhor explicação do efeito diagnosticado, isso quer dizer que o componente culpado pelo problema é o de configuração.

A última fase é o diagnóstico de saída do Netmedic. O diagnóstico recebe como entrada o intervalo de análises em janelas de um minuto e um intervalo de referência histórica. Para calcular a anormalidade de um componente são utilizados os valores que a variável assumiria com base aproximada na distribuição normal, ou seja, para cada variável, a anormalidade de um componente é a anormalidade máxima entre as variáveis. O passo seguinte é o cálculo do peso das arestas. Seja S a fonte e D o destino da aresta de dependência, se S e D estão normais, então dificilmente S está causando algum impacto em D e por isso é atribuído um peso baixo para a aresta. Se ambos S e D estão anormais, então, neste caso, é utilizado o histórico de tempo para analisar este comportamento e calcular o peso da aresta. Neste caso, o histórico de tempo é quebrado em janelas iguais à utilizada como entrada (um minuto) e os valores de S e D são comparados na linha do tempo, se S e D estiverem com problemas ao mesmo tempo no histórico, então existe uma dependência entre S e D e uma aresta com alta dependência é atribuída. Esses pesos nas arestas ajudam a conectar as prováveis causas para os efeitos observados em torno das arestas de maior peso.

Netmedic consegue mapear de forma diferente o funcionamento dos ativos da rede

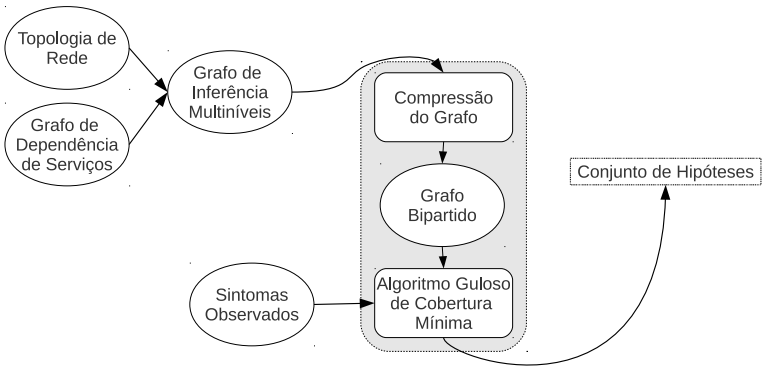


Figura 4.20: *Arquitetura de funcionamento de Spotlight [20].*

utilizando várias variáveis ao invés de uma única informação. É a primeira aplicação voltada para análise de pequenas redes e busca apontar os culpados pelos problemas sem necessariamente conhecer o que está sendo executado na rede. Contudo, a criação dos *templates* citados no artigo exige o conhecimento de como os componentes se comportam e seus relacionamentos. Não ficou claro se estes são genéricos o suficiente para distinguir quaisquer aplicações na rede ou somente o escopo definido na análise de problemas dos chamados coletados. Outro problema é que Netmedic está restrito a poucos *hosts* em rede e, por isso, sua utilização em larga escala é improvável sem que uma atualização ou melhoramentos sejam realizados.

4.7 Spotlight

O sistema de localização de falhas Spotlight [20] é baseado em duas ideias simples. Primeiro, ele comprime o grafo de dependências multiníveis em um grafo bipartido dirigido e com probabilidades nas arestas entre as raízes de problema (servidores, roteadores etc.) e os sintomas (relatórios dos clientes). Depois é executado um algoritmo guloso para fornecer um conjunto de hipóteses das causas dos problemas. De acordo com os autores, o Spotlight chega a ser cem vezes mais rápido que Sherlock nas configurações padrão de execução, mas compromete um pouco a qualidade dos resultados.

A arquitetura de funcionamento do Spotlight é dividida em três componentes, conforme mostra a Figura 4.20. Primeiro, ele usa o grafo de dependência de serviço (GDS) para codificar os relacionamentos de alto nível que existem entre diferentes servidores na publicação de um serviço. O GI é criado utilizando estes relacionamentos de alto nível e a topologia de rede (mesma abstração utilizada pelo Sherlock). Depois, é utilizado um algoritmo para comprimir o GI em um grafo bipartido, simplificando as dependências multicamadas. Por último, é aplicado um algoritmo de cobertura mínima por conjuntos (*Minimum Set Cover*) com peso, para relacionar o conjunto de possíveis candidatos a raízes de problema que melhor explicam os sintomas observados.

O GDS utilizado é igual ao gerado no Sherlock. Os GDSs são computados no Spotlight da mesma forma que no Sherlock, se um serviço B depende de um serviço A, então A

e B vão trocar pacotes entre si. Por essa observação, é calculada a probabilidade de dependência de um serviço A quando do acesso a um serviço B como o número de vezes que o acesso ao serviço B é precedido por um acesso ao serviço A em um tempo pré-definido (intervalo de dependência).

O segundo grafo utilizado é o de inferências. Ele é um grafo direcionado com rótulos representando as dependências entre os componentes de rede. A união dos GDSs e da topologia de rede é utilizada para construir o GI que será utilizado para localização dos problemas. Este grafo possui três tipos de nós:

- os nós raízes de problema, que correspondem aos componentes físicos da rede em que a falha pode afetar o desempenho, os aplicativos;
- os nós de observação, que modelam a experiência do usuário (e são representados pelo par: <cliente, serviço>), ou seja, a percepção sobre o desempenho de um aplicativo (quando ele está funcionando normalmente ou apresentando degradação);
- os nós de ação (diferente de Sherlock que utilizava meta nós), que agem interligando os nós raízes de problema e os nós de observação. Estes últimos nós representam as ações que os serviços de rede têm que tomar para satisfazer a requisição do usuário.

A Figura 4.21 apresenta um trecho do grafo de inferência gerado por um acesso de um cliente a uma página *web*. Este grafo representa um cliente C acessando o serviço S. O nó de observação representa o relatório do cliente, os nós raízes de problema representam os *hosts* onde os serviços estão executando e os nós de ação representam o cliente C acessando o serviço S. Para cada acesso a um serviço diferente é atrelado também um caminho físico representando a ação do cliente acessar o servidor. Isso quer dizer que, ao acessar um servidor, a ação não está limitada somente ao serviço, mas também ao caminho físico até o servidor. Cada serviço adicional que o cliente utilizar na tentativa de acessar a página *web* será adicionado como um nó raiz de problema no GI, ou seja, se estes serviços não estiverem funcionando corretamente, o acesso ao site pode ser comprometido e ele deverá ser considerado o problema no acesso.

Spotlight recebe como entrada o grafo de dependências multinível, mas primeiro o grafo deve ser comprimido em um grafo bipartido com arestas direcionadas. Cada aresta entre um nó raiz de problema e um nó sintoma (similar ao nó de observação no Sherlock) possui peso equivalente à probabilidade do sintoma tornar o nó raiz de problema o nó responsável pela falha. Esta representação simples facilita a utilização do algoritmo de cobertura mínima aplicado que, de acordo com os autores, é melhor do que o método Bayesiano aplicado por Sherlock. Uma vez que o GI é direcionado, acíclico e com pesos nas arestas representando uma distribuição probabilística, é possível utilizar as propriedades Bayesianas do algoritmo de inferência para comprimir o grafo de inferência multiníveis em um grafo bipartido.

A Figura 4.22 mostra visualmente a compressão do GI. Esta compressão funciona de forma simples, pois no grafo bipartido resultante estarão somente os caminhos cujos resultados dos produtos entre as probabilidades dos caminhos sejam os maiores. Por

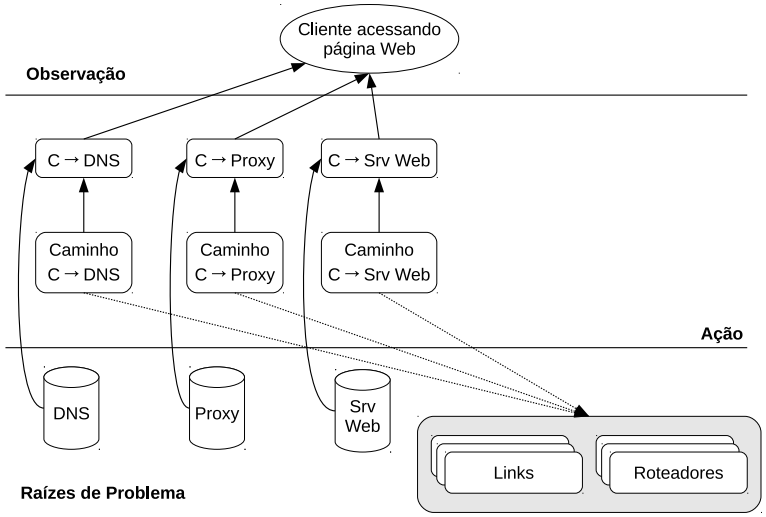


Figura 4.21: *Trecho do GI que representa o acesso de um cliente à uma página web [20].*

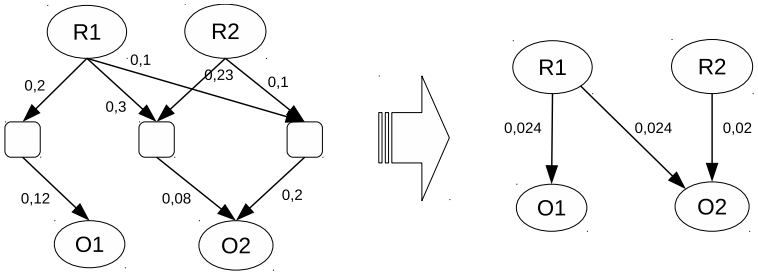


Figura 4.22: *Exemplo de compressão de um grafo de dependências multinível para um grafo bipartido.*

exemplo, partindo de $R1$ há dois caminhos para $O2$, um caminho tem produto $0,3*0,08 = 0,024$ e o outro caminho tem produto $0,1*0,2 = 0,02$, menor que o anterior. Neste caso, o caminho no grafo bipartido será aquele com valor $0,024$. A complexidade de execução deste algoritmo para um grafo completo é $O(|E|) = O(|V|^2)$, mas como na prática este grafo é relativamente esparso, então a complexidade é reduzida para $O(|E|) = O(|V|)$ que é praticamente linear no número de nós no grafo multinível.

A compressão do GI multiníveis fornece o grafo bipartido com os nós raízes de problema R e os nós de observação O. Seja S o conjunto de sintomas observados ($S \subseteq O$). O problema é identificar a hipótese mais provável $H \subseteq R$ que explique S, ou seja, cada membro de S é explicado pela cobertura de pelo menos um membro de H. Note que um sintoma é explicado pela cobertura de um nó raiz de problema se existe uma aresta com valor diferente de zero (probabilidade de dependência) entre o nó raiz de problema e o sintoma. Encontrar H neste caso pode ser modelado como encontrar um conjunto de cobertura para S, em que cada elemento em H pode ser definido como o conjunto de sintomas que foram observados quando este elemento falhou.

Uma vez que o conjunto de sintomas foi definido, o problema é claramente reduzido

a encontrar o conjunto de elementos que explicaria completamente todos os sintomas. Encontrar o conjunto de cobertura mais provável é equivalente a encontrar o menor conjunto de cobertura em que todos os elementos possuem a mesma probabilidade de falhar, levando em consideração que a probabilidade de um grande número de falhas ocorrerem é muito menor do que uma pequena quantidade. Com isso, o problema pode ser reduzido para o problema de encontrar um conjunto de cobertura mínima. Contudo, este problema é NP completo e uma solução linear não é conhecida.

Spotlight utiliza um algoritmo guloso nos pesos para identificar o menor número possível de conjuntos de nós raízes de problema H que explicam os sintomas do conjunto S. Para auxiliar na execução deste algoritmo, são utilizadas duas informações: a *taxa de acerto* dos nós raízes de problema R_i, que é a fração dos nós de observação associados com R_i que aparecem como sintomas e a *relação de cobertura* dos nós raízes de problema R_i, que é a fração do total de sintomas que podem ser explicados por R_i. Com essas informações, o algoritmo de cobertura gulosa inicia com um conjunto de sintomas não explicados igual ao conjunto de sintomas e o objetivo é selecionar os nós raízes de problema que tenham a maior relação de cobertura e taxa de acertos. Uma vez que o nó raiz de problema seja selecionado, os sintomas relacionados a este nó são removidos do conjunto de sintomas não explicados. Quando todos os sintomas forem explicados por algum nó raiz de problema, então o conjunto de hipóteses é retornado.

Spotlight consegue trabalhar de forma eficiente com redes de computadores maiores do que as abordagens anteriores como Sherlock e Netmedic, mas perde qualidade nas soluções encontradas quando comparados os resultados. Seu tempo de execução em relação ao Sherlock é mais rápido e sua precisão aparenta ser melhor do que a do algoritmo Ferret (utilizado no Sherlock). Contudo, apenas observando os testes realizados é possível detectar falhas não explicadas corretamente, como, por exemplo, na análise dos falsos positivos e falsos negativos. Além disso, com duas ou mais anomalias simultâneas Spotlight tem péssimos resultados.

4.8 Comparação Qualitativa dos Métodos Não Intrusivos

Os métodos não intrusivos estudados nas seções anteriores podem ser comparados utilizando algumas métricas. Para ser o mais abrangente possível, serão utilizadas duas métricas quantitativas (com resultados numéricos extraídos diretamente dos trabalhos originais) e uma métrica qualitativa. As métricas quantitativas se referem à completude, precisão e eficiência e a métrica qualitativa se refere à generalidade. Para que o significado destes termos fique mais claro, suas definições seguem abaixo:

- Precisão: é a taxa de acertos de uma abordagem em relação a todas as requisições realizadas;
- Eficiência: reflete a sobrecarga que uma abordagem adiciona a uma aplicação ou sistema;

- Generalidade: está relacionado com configurações de hardware e software em que uma abordagem é aplicável, incluindo fatores como linguagens de programação, sincronização de relógio, presença ou ausência de *logs* de sistema, modelo de paralelismo etc.;

Tomando como base esses parâmetros de comparação, a Tabela 4.1 compara os sete métodos não intrusivos abordados neste trabalho. Esta tabela não cobre todos os aspectos de todas as ferramentas e métodos, mas apresenta um modelo razoável de comparação.

Tabela 4.1: *Tabela qualitativa dos métodos não intrusivos avaliados neste trabalho.*

	Precisão	**Eficiência**	**Generalidade**
Project5	depende da quantidade de fluxos	*nesting* - $O(nlogn)$ *convolution* - $O(\frac{t}{s}log\frac{t}{s})$	dependente do relógio
Wap5	não informado	$O(nlogn)$	biblioteca de captura LibSockCap
Sherlock	95% $(k = 1, obs = 40)$	$O(2 * r)^k, k \in \{1, 2, 3, 4\}$	estados como ativo, problemático ou inativo
eXpose	não se aplica	$O(\sum_{w=0}^{W} S_w^2)$	quantidade de k fluxos simultâneos
Constellation	96% (recall=94%)	$< 1\%$	winpcap/tcpdump, strace, ptrace ou ETW
Netmedic	80%	1%	informações de aplicação, *firewall*, etc.
Spotlight	$96,5\%$ $(k = 1, obs = 40)$	$O(\|V\|^2)$	estados como ativo, problemático ou inativo

A dificuldade em mapear as características de cada trabalho é considerável, pois, embora todas as abordagens sejam não intrusivas e detectem anomalias, cada uma delas mapeia o problema de forma diferente. Isso só não acontece quando um trabalho é derivado ou comparado com um anterior, como no caso do Spotlight com o Sherlock. Além disso, mesmo quando outros trabalhos citam ou comparam resultados, abordagens diferentes complicam o mapeamento de pontos de comparação.

É possível comparar a Tabela 4.1 deste capítulo com a Tabela 3.1 do capítulo anterior. No total foram definidas quatro métricas (precisão, eficiência, generalidade e transparência) mas que nem sempre fazem sentido quando contextualizado o conjunto de abordagens em estudo. Por exemplo, os métodos intrusivos são precisos e extraem, no geral, corretamente todos os caminhos das requisições, por isso, não faz sentido comparar esta métrica entre os métodos intrusivos. O mesmo se aplica à métrica de transparência para os métodos não intrusivos, pois, por definição, estes métodos não devem influenciar o ambiente que está sendo analisado, embora possam realizar coletas de dados específicos.

O vPath sugere uma quinta métrica: a completude, que representaria a capacidade de uma ferramenta ou abordagem extrair corretamente os caminhos de uma requisição precisamente. Contudo, a maioria dos trabalhos não informam ou abordam esta informação de forma precisa, tornando-a uma métrica muito específica para o próprio vPath.

4.9 Outros Métodos Não Intrusivos

Os métodos não intrusivos apresentados neste capítulo fornecem uma visão geral do problema e das diferentes técnicas que podem ser utilizadas para resolvê-lo. Além dos métodos já detalhados, há vários outros na literatura que atacam o mesmo problema. As subseções a seguir apresentam métodos mais recentes que também foram estudados ao longo deste trabalho mas que são variações dos já detalhados neste capítulo. Além disso, existem outras abordagens diferentes mas que não agregam informações pertinentes à solução desenvolvida.

4.9.1 Orion

Orion [8] é uma ferramenta não intrusiva de detecção de anomalias. Seu diferencial é descobrir as dependências de aplicações na rede utilizando os cabeçalhos dos pacotes e informações de tempo de acesso. A diferença na análise de tempo do Orion em relação a outros trabalhos está relacionada aos picos de atrasos nos acessos. Esses picos são extraídos de histogramas de tempos de acesso, em que cada distribuição de tempo é tratada como um sinal. Com essas informações o Orion utiliza filtros de ruído baixo para reduzir ruídos aleatórios e decompõe os sinais em um espectro de frequências utilizando a Transformada Rápida de Fourier (FFT - *Fast Fourier Transform*) para auxiliar na extração das dependências de serviços.

Os trabalhos mais próximos do Orion são Sherlock [2] e eXpose [22]. O Orion critica o fato destes dois trabalhos anteriores necessitarem do tráfego de coocorrência (repetições de um mesmo tipo de tráfego) para extrair dependências. Além disso, ambas as abordagens anteriores dependem de uma janela de tempo. O maior problema, de acordo com Orion, em determinar uma janela fixa de tempo é a dificuldade em equilibrar a quantidade de falsos positivos e falsos negativos quando analisadas redes diferentes.

4.9.2 DYSWIS

DYSWIS [38] (*Do you see what I see*) é uma ferramenta criada para diagnosticar problemas na rede usando histórico de informações e testes ativos. Diferente do Sherlock [2] e do Spotlight [20], que utilizam dependências e informações da topologia da rede para construir um GI utilizado na detecção de anomalias, o DYSWIS trata cada nó da rede como uma fonte em potencial de gerenciamento capaz de coletar informações sobre o funcionamento da rede. Desta forma, cada nó possui uma visão da rede que é agregada em uma visão geral. O que DYSWIS faz é correlacionar estes nós com um histórico de informações de falhas. Uma vez que um nó diagnostica uma falha, se outros sistemas estiverem experimentando problemas similares, estas informações são combinadas com conhecimento local para tentar estimar as origens dos problemas.

Ao invés de extrair as dependências como os outros modelos não intrusivos fazem, DYSWIS utiliza um sistema baseado em regras. Cada regra representa uma dependência

entre vários componentes de rede. Esse motor de regras (DROOLS) codifica as relações de dependência utilizando sondas ou testes e consultas. Dessa forma, um conjunto de nós DYSWIS é capaz de realizar inferências sobre uma falha em particular, realizando diagnósticos e testes baseando-se em dependências entre componentes de rede e protocolos. A principal contribuição do DYSWIS é a agregação das informações da visão da rede coletada por seus nós com o histórico de informações de falhas.

4.9.3 SNAP

SNAP [43] é uma ferramenta escalável criada para analisar problemas de desempenho em *data centers*. Seu objetivo é fornecer uma ferramenta leve, genérica e precisa no diagnóstico de problemas de desempenho em ambientes distribuídos. O SNAP pode ser comparado a ferramenta intrusiva DAPPER [37], mas diferente desta, SNAP utiliza coleta de estatísticas do TCP, informações de *logs* de chamadas de *sockets* e as correlaciona com recursos compartilhados (enlaces, *switches*, roteadores, etc.) para determinar a origem do problema. Além disso, para que SNAP funcione corretamente são utilizadas duas propriedades existentes em *data centers* modernos: o conhecimento total da topologia da rede (pilha de rede, configurações e o mapeamento das aplicações e seus servidores) e a possibilidade de analisar a pilha de rede para observar a evoluções de conexões TCP diretamente ao invés de tentar inferir o comportamento de pacotes TCP em traços de rede. Dessa forma, ele consegue identificar rapidamente a localização (servidor, enlace, *switch*, etc.) correta e a camada de rede (aplicação, rede, etc.) correta, de um problema.

4.9.4 NSDMiner

NSDMiner [30] é uma ferramenta não intrusiva que descobre dependências de rede analisando passivamente o tráfego de rede. Ele separa as dependências em dois tipos: local-remoto e remoto-remoto. As dependências local-remoto ocorrem quando um dispositivo, para prover um serviço local acessa um serviço remoto (um servidor web que depende do servidor de banco de dados). As dependências remoto-remoto ocorrem quando para acessar um serviço remoto é necessário acessar antes outro serviço remoto (um navegador de internet que para acessar uma página depende do acesso ao DNS). NSDMiner concentra seu trabalho na melhoria e eficiência de descobertas de dependências local-remoto.

NSDMiner utiliza a mesma estrutura lógica do Project5 [1], encadeando requisições baseando-se no tempo em que ocorreram nos traços de rede. A principal diferença do NSDMiner é que não existe a dependência local do tempo como no Project5 (tempo dos dispositivos de rede), ao invés disso, o tempo em que a requisição aconteceu é utilizado como janela para acompanhar as dependências. Por isso, ele consegue agregar dependências de acessos apenas analisando a cadeia de requisições em um serviço. Além disso, ele só analisa as dependências local-remoto que são mais comuns em servidores de conteúdo sem cobrir o outro tipo de dependência (remoto-remoto), o que limita o escopo do trabalho. Por isso, a principal contribuição do NSDMiner é exatamente a melhoria na detecção automática de dependências do tipo local-remoto de serviços de rede.

4.10 Considerações Finais

Este capítulo apresentou as técnicas não intrusivas de detecção de anomalias em ambientes distribuídos. As técnicas, embora diferentes, utilizam uma ideia comum que é a inferência de anomalias usando apenas o tráfego coletado da rede. As variações de cada técnica tentam resolver os problemas inerentes ao modelo de inferência estatística, como a baixa ocorrência de acessos a serviços esporádicos e falsos positivos.

As técnicas não intrusivas são melhores que as intrusivas em ambientes onde existem sistemas legados, ferramentas que podem evoluir sem o controle dos administradores de rede e em locais com pouca documentação sobre a organização topológica da rede. Por esses motivos, ferramentas genéricas desenvolvidas para auxiliar administradores em redes corporativas tendem a adotar técnicas similares às exploradas neste capítulo. No próximo capítulo é descrita a ferramenta desenvolvida neste trabalho. A ferramenta foi baseada principalmente em dois trabalhos não intrusivos descritos neste capítulo, Sherlock e Spotlight.

Capítulo 5

Ferramenta de Detecção de Anomalias Nemo

A ferramenta Nemo foi inspirada na ferramenta Sherlock e utiliza boa parte de sua modelagem conceitual, bem como parte das estruturas de dados que foram inicialmente introduzidas em [2]. Entretanto, Nemo introduz novos conceitos teóricos e algoritmos bem mais eficientes que os utilizados por Sherlock. Assim como Sherlock, a ferramenta Nemo é dividida em dois componentes principais: agente e gerente. A organização geral de Nemo está ilustrada na Figura 5.1.

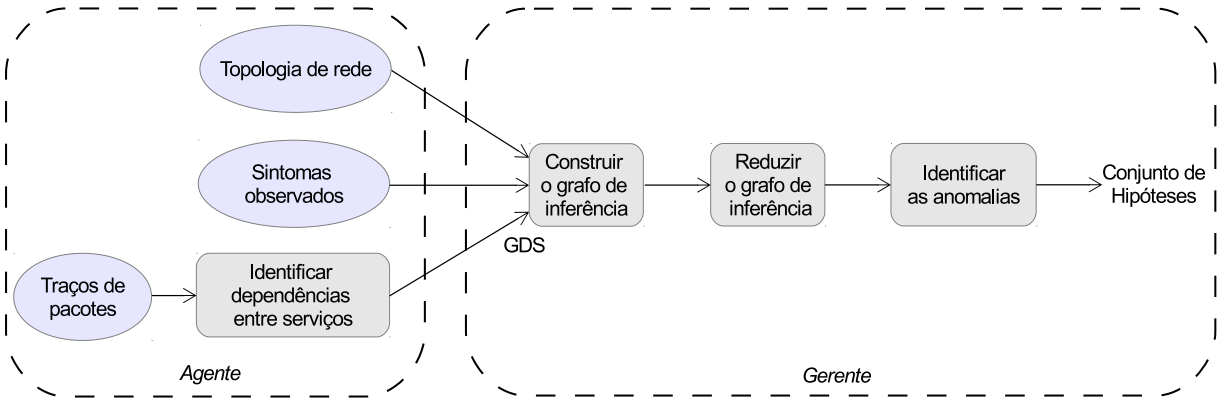


Figura 5.1: *Visão geral da ferramenta Nemo.*

Os agentes de Nemo são instalados em estações de trabalho e monitoram os pacotes que são enviados e recebidos pelas estações. Este é o processo normal de captura de pacotes de Nemo, mas não o único. Nemo poderia utilizar pacotes coletados diretamente de portas clonadas (*mirroring ports*) de roteadores ou *switches*, por exemplo. A partir dos pacotes monitorados, um agente constrói localmente um Grafo de Dependências de

Serviços (GDS) para os serviços acessados pela estação e monitora os tempos de resposta de cada acesso para construir estatísticas confiáveis sobre o tempo médio de acesso de cada serviço. Quando o tempo de resposta é muito superior ao tempo médio, indicando possivelmente uma anomalia, os agentes solicitam ao gerente que seja executado o processo de diagnóstico de anomalias. Além disso, os agentes coletam informações de topologia da rede, por meio de um módulo de `traceroute` implementado internamente nos agentes, e enviam essas informações para o gerente.

O gerente é responsável por combinar todas as dependências entre serviços fornecidas pelos agentes e consolidar as visões locais em um GDS global para toda a rede. Além disso, o gerente faz a união do GDS com as informações de topologia para gerar um Grafo de Inferência (GI). O GI captura todas as dependências entre serviços e elementos de rede como enlaces, roteadores e servidores. O gerente também executa o processo de diagnóstico de anomalias quando solicitado pelos agentes. A Seção 5.1 descreve mais detalhadamente como os GDSs são construídos por Nemo.

As próximas seções introduzem conceitos teóricos e descrevem as principais estruturas de dados utilizadas neste trabalho.

As principais diferenças entre Nemo e Sherlock estão nos módulos de redução do GI (inexistente no Sherlock) e no módulo de identificação de anomalias. Embora a ferramenta tenha exigido um grande esforço de implementação e implantação em um ambiente de produção, o foco deste trabalho está nas melhorias teóricas e algorítmicas introduzidas nos módulos de redução do GI e na detecção de anomalias. Esses módulos serão discutidos nas Seções 5.4 e 5.5, respectivamente. A Seção 5.2 explica como é construído o GI e a Seção 5.3 o conceito geral de Redes Bayesianas. Os aspectos mais relevantes da implementação da ferramenta serão discutidos na Seção 5.6.

5.1 Grafo de Dependências de Serviços

O Grafo de Dependências de Serviços (GDS) descreve as dependências existentes entre serviços. Cada serviço acessado por um cliente, e capturado pelos agentes de Nemo, é mapeado para um nó do GDS. Um serviço S_1 é dependente de um serviço S_2 se o acesso a S_1 é precedido por um acesso a S_2. O GDS representa essa dependência com uma aresta direcionada de S_2 para S_1 e atribui um peso (probabilidade) a essa aresta de acordo com a força da dependência. A probabilidade de dependência (peso da aresta) é calculada dividindo-se o número de vezes que acessos ao serviço S_2 precedem acessos a S_1 dentro de um intervalo limite de tempo (intervalo de dependência), pelo número de vezes que S_1 é acessado em todos os traços coletados pelos agentes. O intervalo de dependência é um parâmetro do sistema e pode ser alterado de acordo com características específicas do sistema que está sendo monitorado. O valor sugerido por Sherlock e também utilizado neste trabalho é fixo e igual a 10ms.

Esse valor se mostrou bastante eficaz na captura das dependências. Contudo, como o valor é estático, ele pode gerar falsos positivos. Em Sherlock e, consequentemente em Nemo, a estratégia para evitar alguns problemas com falsos positivos, foi utilizar

uma heurística simples de coocorrência. Esta heurística consiste em utilizar o cálculo do intervalo médio de acesso a um mesmo serviço (I) e estimar a chance de coocorrência como $10ms/I$. As dependências que tiverem as probabilidades de dependência superiores a esse limiar são enviadas ao gerente para agregação dos dados, caso contrário, a informação é apenas armazenada localmente no agente até atingir um valor superior à chance de coocorrência.

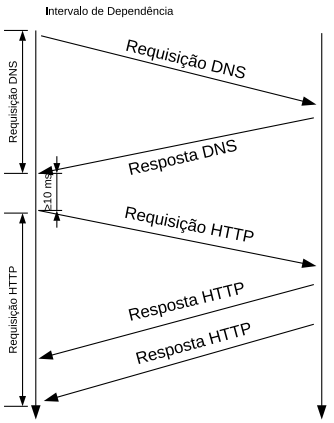


Figura 5.2: *Mapeamento de dependências entre serviços.*

A Figura 5.2 ilustra o mapeamento da dependência entre os serviços HTTP e DNS. Nesse caso, como o acesso ao serviço HTTP ocorre dentro do intervalo de dependência do acesso ao serviço DNS, infere-se que o serviço HTTP é dependente do serviço DNS. A probabilidade da dependência é calculada de acordo com o número de vezes que esses dois acessos ocorrem dentro do intervalo de dependência. Observe que essa maneira de se calcular as probabilidades de dependência captura adequadamente situações em que as respostas de DNS são armazenadas em cache e que acessos seguidos ao mesmo endereço não precisam de novos acessos ao servidor de nomes. Nesse caso, a probabilidade de dependência será menor que 1,0.

A forma como Sherlock, Spotlight e Nemo capturam as informações na rede esbarra em um problema inerente a técnicas não intrusivas, que é a baixa quantidade de amostras para extração de dependências. Durante a construção do GDS, para uma extração concisa de dependências, é necessário um número mínimo de ocorrências de uma dependência. Nemo soluciona este problema para um serviço com baixa quantidade de amostras enviando, depois de um certo tempo (10 minutos), o serviço com sua lista de tempos de acesso para que o gerente possa definir uma média ativa e problemática global até que o agente possua informações suficientes para manter suas próprias médias locais. Por outro lado, o Sherlock utiliza uma abordagem mecânica, introduzindo robôs de repetição que imitam ações do usuário até alcançar o número mínimo de amostras necessárias à geração da dependência. Além disso, ele agrega informações de vários agentes de forma global para reduzir o número de falsos positivos e gerar, como em Nemo, uma média global. Nemo não utiliza em sua implementação robôs de mimetização de ações, embora esta técnica tenha sido utilizada como forma de acelerar a coleta de informações da rede e geração das modas ativas e problemáticas nos agentes em alguns experimentos.

5.2 Grafo de Inferência de Rede

Um Grafo de Dependência de Serviços (GDS) captura adequadamente as dependências entre serviços, mas não inclui informações sobre a infraestrutura de comunicação. Um Grafo de Inferência (GI) estende um GDS e é definido como um Grafo Direcionado Acíclico (GDA), com pesos nas arestas, que representa as dependências entre componentes de um ambiente distribuído, sendo que componentes incluem serviços e elementos de hardware. Em um GI, há três tipos de nós:

- **nós de observação:** modelam a experiência que o usuário está observando ao acessar um serviço. Essa experiência representa o tempo de acesso ao serviço e pode indicar que o serviço está *ativo* (operando normalmente e com tempos de resposta baixos), *problemático* (operando, mas com tempos de resposta muito altos), ou *inativo*;
- **nós raízes de problema:** representam servidores, equipamentos de rede, enlaces ou serviços. Quando um nó desse tipo apresenta um problema, alguns nós de observação acusarão o problema, fazendo com que o usuário perceba a degradação de um serviço;
- **meta nós ou nós de interligação:** servem para interligar os nós de observação a outros nós do GI que podem ser causadores de problemas. Eles são de três tipos: ruído máximo (ligações um-para-um), seleção (modelam balanceadores de carga) e *failover* (modelam redundância de servidores do tipo mestre/escravo).

O diferencial da modelagem realizada por Sherlock [2] em relação a outras abordagens, como Shrink [23] e SCORE [25], foi a adição do estado *problemático* em cada nó. Cada nó do GI possui um estado que é representado pela probabilidade do nó estar *ativo*, *problemático* e *inativo*. Dessa forma, um nó com estado $(0,3; 0,5; 0,2)$ indica que o nó possui 30% de chance de estar *ativo*, 50% de chance de estar *problemático* e 20% de chance de estar *inativo*. A soma das probabilidades $P_{\text{ativo}} + P_{\text{problemático}} + P_{\text{inativo}}$ deve ser $1,0$.

Assim como no GDS, as arestas no GI são direcionadas e possuem pesos (probabilidades) que representam a força da dependência entre dois nós. Para cada tipo de meta nó, há uma tabela-verdade que indica como as probabilidades de um nó são calculadas a partir das probabilidades de seus antecessores e dos pesos das arestas. As tabelas verdade para cada tipo de nó estão definidas originalmente em [2] e podem ser encontradas neste trabalho na Subseção 4.3.

Finalmente, o GI possui dois nós especiais utilizados para possíveis erros no modelo, o nó *sempre problemático* (AT - *always troubled*) e o nó *sempre inativo* (AD - *always down*), que possuem seus estados definidos como $(0; 1; 0)$ e $(0; 0; 1)$, respectivamente. Esses nós são vinculados a cada nó de observação para representar possíveis informações que não estão mapeadas no modelo.

A Figura 5.3 mostra parte de um GI gerado automaticamente por Nemo. Os roteadores e os enlaces foram agregados para reduzir a quantidade de arestas e facilitar a visualização.

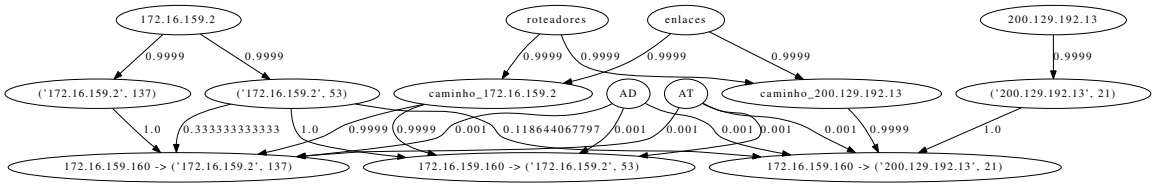


Figura 5.3: *Parte de um GI de uma rede em produção gerado automaticamente por Nemo.*

Como o GI foi gerado automaticamente algumas informações como, por exemplo, o tipo de nó (nó de observação, meta nó e raiz de problema) não ficaram visíveis, mas são diferenciados na estrutura interna de Nemo. As arestas do GI representam a força de dependência entre os nós de observação, os meta nós e os nós raízes de problema.

Esse trecho do GI foi construído com informações coletadas por um agente instalado na rede da UFMS. O grafo mostra, entre outras coisas, as dependências dos serviços NetBios (137) e do serviço DNS (53) quando realizado o acesso ao serviço de FTP (21) (para simplificar, os roteadores e enlaces foram agregados na imagem).

5.3 Redes Bayesianas

Redes Bayesianas são comumente utilizadas para realizar inferências probabilísticas em áreas como Estatística, Inteligência Artificial, Mineração de Dados etc. Uma Rede Bayesiana (RB) nada mais é que um modelo gráfico que codifica relações probabilísticas entre variáveis de interesse. Sua utilidade advém de sua capacidade de codificar dependências entre todas as variáveis de interesse, de lidar com situações em que nem todas as informações são conhecidas, e de ser útil para se inferir relacionamentos causais. Segue, abaixo, a definição formal de uma Rede Bayesiana.

Definição 1 *Uma Rede Bayesiana* $\mathfrak{B}$ *é definida como um par* $\mathfrak{B} = (G, P)$*, em que* $G = (V(G), A(G))$ *é um grafo direcionado acíclico com conjunto de vértices* $V(G) = \{X_1, \ldots, X_n\}$ *e conjunto de arestas* $A(G) \subseteq V(G) \times V(G)$*, e* P *é uma distribuição de probabilidade conjunta definida nas variáveis correspondentes aos vértices do conjunto* $V(G)$ *como segue:*

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | \pi(X_i))$$

em que $\pi(X_i)$ *representa o conjunto de pais (ancestrais diretos) de* X_i*.*

Uma grande contribuição de Sherlock foi a modelagem do problema de diagnóstico de anomalias como um GI. O GI pode ser visto como uma RB, pois é um grafo direcionado acíclico em que os nós representam variáveis aleatórias cujos valores são determinados pelos seus ancestrais diretos de acordo com uma tabela-verdade. Sendo assim, o problema de deteção de anomalias em Sherlock fica reduzido ao problema de se explicar inteiramente

as variáveis desconhecidas na RB, ou seja, descobrir as probabilidades dos nós raízes de problema que melhor explicam as probabilidades dos nós de observação.

5.3.1 *d*-separação e *d*-conexão

Por tentar explicar todas as variáveis desconhecidas da RB, Sherlock é bastante ineficiente, tanto do ponto de vista de armazenamento quanto do ponto de vista de tempo de execução. Nemo tira vantagem do domínio do problema e tenta explicar somente as variáveis que interferem diretamente em sintomas reportados pelos usuários. Para isso, Nemo explora a propriedade de independência condicional em RBs para descartar variáveis que sejam de fato independentes das variáveis de interesse e que, portanto, não interferem em seus valores. A propriedade de independência condicional em RBs foi bastante estudada no passado, resultando no conceito de *d*-separação. Geiger *et al.* [15] propuseram um algoritmo bastante eficiente, de complexidade linear, para o cálculo da *d*-separação em uma RB.

O conceito de *d*-separação é melhor entendido quando associado ao conceito de *d*-conexão. A definição desses conceitos utiliza os termos nós colisores e nós não colisores. A função desses nós pode ser melhor compreendida pelas representações gráficas mostradas na Figura 5.4. Nós colisores e não colisores são nós cujos graus de saída são zero e diferente de zero, respectivamente.

Figura 5.4: *Nó v_1 como (a) colisor e (b) não colisor.*

Definição 2 ***d*-conexão:** *Seja G um grafo direcionado em que X, Y e Z são conjuntos disjuntos de vértices de G, então X e Y são d-conectados por Z em G, se, e somente se, existe um caminho não direcionado U entre vértices de X e de Y, tal que para cada nó colisor c em U, c ou um descendente de c está em Z, e nenhum nó não colisor pertence a Z.*

Definição 3 ***d*-separação:** *X e Y são d-separados por Z em G, se, e somente se, eles não são d-conectados por Z.*

As Definições 2 e 3 formalizam os conceitos de *d*-conexão e *d*-separação respectivamente. A Figura 5.5 mostra uma simplificação visual da regras descritas na definição de *d*-separação. Se existe caminho não direcionado entre os nós do conjunto X e Y e, se este caminho for bloqueado pelos nós de Z (nós seguindo a regra "a"), então X e Y são *d*-separados. Senão, se existe um nó v no caminho e este nó (seguindo a regra "b") e seus descendentes não pertencem a Z, então X e Y também são *d*-separados. Caso contrário,

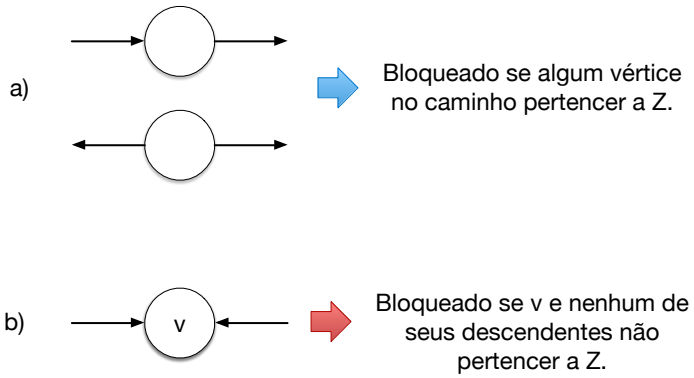


Figura 5.5: *Regras práticas de* d*-separação extraídas da definição.*

X e Y não são d-separados. A Figura 5.6 mostra dois conjuntos Y e R em que Y é d-separado de X por Z e R não é d-separado de X por Z.

O Algoritmo 1 apresenta um pseudo código eficiente de d-separação de conjuntos em um grafo. No Apêndice A está a implementação em Python do algoritmo da d-conexão, que pode ser convertido facilmente na d-separação. Na Seção 5.4, esses conceitos serão utilizados no desenvolvimento de um novo algoritmo que reduz significativamente os GIs gerados por Nemo.

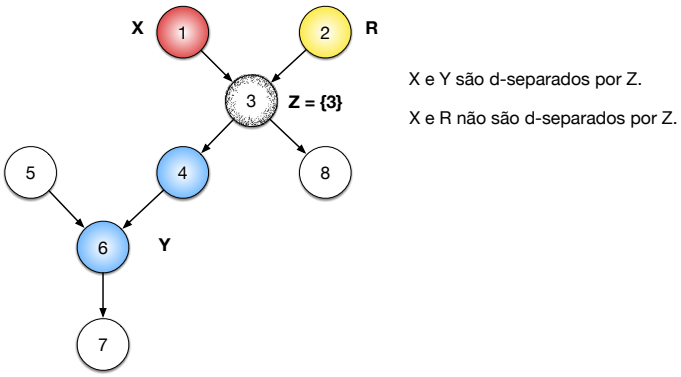


Figura 5.6: *Exemplo de conjuntos* d*-separados.*

Algoritmo 1: d-separação (G, X, Z)

Entrada: Grafo G e dois conjuntos de nós X e Z
Saída: Conjunto $R \subseteq V(G)$

```
1  início
2      R = {};
3      L = {};
4      L.add(X, ↑);
5      visitados = {};
6      enquanto |L| ≠ 0 faça
7          y, d ← L.pop();                          // y, d = (nó y, direção d)
8          se (y, d) ∈ visitados então continua;
9          visitados.add((y, d));
10         se y ∉ Z então R.add(y);
11         se d = ↑ ∧ y ∉ Z então
12             L.add(filhos(y), ↓);
13             L.add(pais(y), ↑);
14         senão se d = ↓ então
15             se y ∉ Z então L.add(filhos(y), ↓);
16             se y ∈ Ancestrais(Z) então L.add(pais(y), ↑);
17     fim
18     R.add(G - Z) ;                          // apenas os nós não alcançáveis
19     retorna R;
20 fim
```

5.4 Redução do Grafo de Inferência

O GI, como proposto por Sherlock, inclui todos os serviços, elementos de hardware e clientes monitorados em uma rede corporativa. Em situações de anomalias, apenas parte desses componentes realmente apresentam problemas. Além disso, dependendo da anomalia, poucos clientes são de fato afetados. Como exemplo, pode-se citar um defeito em um roteador que afeta apenas os clientes das sub-redes roteadas pelo equipamento defeituoso.

Com base nas observações acima, Nemo procura identificar apenas os nós do GI que influenciam os sintomas observados e descartar os demais nós que são comprovadamente independentes dos sintomas. Para se descartar os nós com a certeza de que eles não são os causadores dos sintomas, Nemo utiliza o conceito de d-separação (Definição 3).

A questão chave para utilização da d-separação para redução do GI é a definição de pelo menos dois dos conjuntos X, Y e Z, pois o terceiro é obtido de modo eficiente (em tempo linear) pelo Algoritmo 1 de d-separação. Nemo constrói o conjunto Z como sendo os nós de observação que estão reportando problemas, ou seja, os nós sintomas. O conjunto X é definido como sendo os nós raízes de problema que são antecessores dos nós de Z, pois esses são nós que podem afetar os nós sintomas. Os nós de X podem ser obtidos utilizando um algoritmo de busca em largura a partir dos nós sintomas invertendo-se as orientações das arestas. O Algoritmo 2 sumariza essa ideia. Com X e Z definidos, o

conjunto Y, obtido pela d-separação (Linha 4), contém os nós do GI que são independentes condicionalmente dos sintomas. Para se obter um grafo reduzido R, os nós de Y e suas arestas são removidas do grafo original. R conterá os nós de X, os nós de Z, os nós de observação que são dependentes condicionalmente de Z que podem explicar os sintomas observados, e os meta nós que unem os demais nós do grafo.

Algoritmo 2: Redução (G)

Entrada: Grafo de Inferência G.
Saída: Subgrafo R de G com apenas os nós e arestas que são relevantes para explicação dos sintomas.

1 **início**
2 $Z \leftarrow$ Conjunto de nós de observação de G que estão observando degradações nos acessos (sintomas);
3 $X \leftarrow$ Conjunto de nós raízes de problema de G que são ancestrais dos nós de Z;
4 $Y \leftarrow$ d-separação(G, X, Z); `/* Y contém os nós de G tal que, X e Y são d-separados por Z em G. */`
5 $R \leftarrow G - Y$;
6 **retorna** R;
7 **fim**

5.5 Função de Pontuação

O processo de detecção e diagnóstico de anomalias em um GI no Algoritmo 3 (Ferret) e também em Nemo, consiste em encontrar um vetor de atribuição de probabilidades aos estados *ativo, problemático* e *inativo* dos nós raízes de problema que melhor explique os estados observados pelos nós de observação. A ideia básica é propagar os estados (probabilidades) dos nós raízes de problema até os nós de observação e comparar os valores propagados com os estados dos nós de observação obtidos por meio das evidências. As evidências nada mais são que os tempos de acesso aos serviços. Para se determinar o melhor vetor, o algoritmo de inferência utiliza uma função de pontuação que leva em consideração os tempos de resposta dos serviços monitorados pelos agentes. A diferença em relação ao Nemo está na função de pontuação adotada. Enquanto Sherlock utiliza a função de distribuição de probabilidades normal, Nemo utiliza a função de pontuação

descrita no Algoritmo 4.

Algoritmo 3: Ferret (Observações O, Grafo de Inferência G, int X)

```
   // vetores de atribuição para os nós raízes do problema com pelo
      menos k anormais em um dado momento
 1 Candidatos ← (ativo, problemático, inativo);
   // Lista que irá armazenar as X melhores pontuações dos vetores de
      atribuição
 2 Lista_X ← ();
 3 para cada R_a ∈ Candidatos faça
 4 |   Atribua os Estados para todos os nós Raízes do Problema em G como R_a;
 5 |   Pontuação(R_a) ← 1;
 6 |   para cada Nó n ∈ G faça
 7 |   |   Compute P(n) dado P(pais de n);
 8 |   fim
   |   // Pontuando os nós de Observação
 9 |   para cada Nó n ∈ G faça
   |   |   /* Quão bem o R_a atribuído explica a observação em n?  - o
   |   |      Ferret utiliza, originalmente, a função de Distribuição de
   |   |      Probabilidade Normal.  No Nemo discretizamos está função  */
10 |   |   s ← P( evidência(n)| probabilidade de densidade de n);
   |   |   // Pontuação Final
11 |   |   Pontuação(R_a) ← Pontuação(R_a) ∗s;
12 |   fim
13 |   Inclua o R_a na Lista_x se a Pontuação(R_a) estiver entre as X maiores;
14 fim
15 retorna Lista_x
```

Em [2], foi observado que os tempos de resposta dos serviços monitorados seguem uma distribuição bimodal em que uma moda caracteriza o tempo médio quando o serviço está normal (*ativo*) e uma segunda moda que caracteriza o tempo médio quando o serviço está enfrentando problemas (*problemático*). Nemo monitorou alguns serviços da rede da UFMS e constatou que a hipótese de distribuição bimodal é de fato verdadeira. A Figura 5.7 mostra a distribuição de tempos de um dos serviços monitorados. As amostras próximas às modas, quando consideradas separadamente, formam distribuições normais com médias bem próximas às modas. Nemo isola as duas distribuições utilizando o método de clusterização "*k-means*" para encontrar as duas modas e utiliza o ponto médio entre as duas modas para separar as distribuições (ativa e problemática).

Nemo e Sherlock diferem em suas funções de pontuação, pois Sherlock assume e utiliza diretamente a Função de Distribuição de Probabilidade Normal para pontuar um tempo de acesso observado. Isso faz com que tempos de acesso menores que a média recebam pontuações inferiores ao tempo médio ou, até mesmo, a tempos ligeiramente superiores à média. Nemo, por outro lado, explora novamente conhecimento específico do domínio do problema para propor uma nova função de pontuação. A função de pontuação de Nemo é discreta e explora a ideia de que tempos de acesso inferiores à média ou ligeiramente

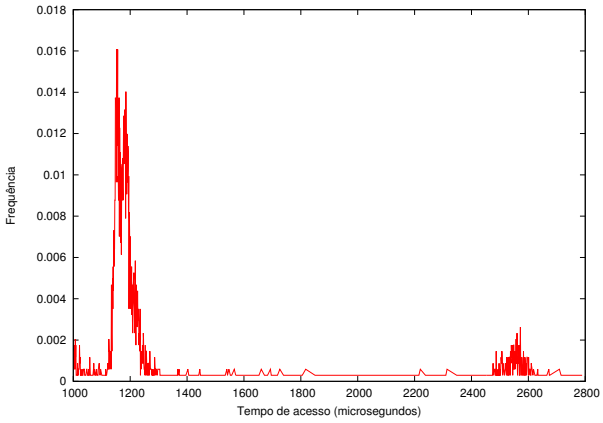


Figura 5.7: *Distribuição de tempos de resposta de um serviço monitorado por Nemo. A primeira moda indica que o serviço está ativo e a segunda problemático.*

superiores devem ter pontuações iguais, pois um tempo de acesso inferior à média indica que o serviço está operando normalmente. Nemo considera apenas os tempos de acesso válidos, de modo que tempos muito pequenos, como os decorrentes de *TCP reset*, são descartados. Os resultados de simulação apresentados na Seção 6.3 mostram que essa função de pontuação aumenta a precisão do algoritmo de inferência nos casos avaliados.

Algoritmo 4: Pontuação (n)

Entrada: Nó de observação n.
Saída: Pontuação em n.

início
  P_{ativo}, $P_{\text{problemático}}$, $P_{\text{inativo}} \leftarrow$ Estado de n;
  μ_{ativo}, $\sigma_{\text{ativo}} \leftarrow$ Média e desvio padrão da distribuição *Ativo* de n;
  se $evidência(n) < \mu_{ativo} + \sigma_{ativo}$ **então**
    retorna $P_{\text{ativo}} * 1,0$;
  se $evidência(n) < \mu_{ativo} + 2 * \sigma_{ativo}$ **então**
    retorna $P_{\text{ativo}} * 0,7 + P_{\text{problemático}} * 0,25 + P_{\text{inativo}} * 0,05$;
  se $evidência(n) < \mu_{ativo} + 3 * \sigma_{ativo}$ **então**
    retorna $P_{\text{ativo}} * 0,4 + P_{\text{problemático}} * 0,5 + P_{\text{inativo}} * 0,1$;
  se $evidência(n) < \mu_{ativo} + 4 * \sigma_{ativo}$ **então**
    retorna $P_{\text{ativo}} * 0,05 + P_{\text{problemático}} * 0,75 + P_{\text{inativo}} * 0,2$;
  retorna $P_{\text{problemático}} * 0,8 + P_{\text{inativo}} * 0,2$;
fim

5.6 Aspectos de Implementação

A ferramenta Nemo, composta de agentes e um gerente, foi implementada e testada na rede de produção da UFMS. Os agentes de Nemo foram desenvolvidos utilizando a linguagem multiplataforma Python na versão 2.6.6 com módulos funcionais em ambientes

Windows e Linux (netifaces, *socket*, pcapy, etc.). Além disso, em cada sistema operacional, foi necessário utilizar uma biblioteca de captura de pacotes de rede, em Windows a WinPcap (que precisou ser instalada) e em Linux a LibPcap (nativa do sistema).

Os agentes de Nemo foram instalados em 20 estações de trabalho Windows. A implantação desses agentes foi feita utilizando a tecnologia de Políticas de Grupo (*Group Policy*) do AD (*Active Directory*) em execução no servidor de domínio com Windows Server 2008. Para a realização da instalação distribuída via AD, foi criado um pacote executável MSI (*Microsoft Software Installer*), exigência do ambiente, utilizando o *framework* py2exe de Python.

O gerente foi desenvolvido em Python e C++, de tal forma que a parte de comunicação e a interface foram desenvolvidas em Python e a parte de processamento em C++. Essa separação foi necessária devido ao enorme *overhead* existente em Python e que tornava o processamento extremamente lento. Härkönen [17] fez um comparativo entre a execução pura de Python, Python embarcado com C e C puro e mostrou que a implementação em C é, pelo menos, 30 vezes mais rápida que a implementação feita somente em Python.

Além disso, testes locais utilizando uma implementação alternativa do interpretador Python, o PyPy [35] (que utiliza estratégias de compilação *Just-in-Time*), mostraram que ao substituir o interpretador padrão pelo PyPy na execução do Nemo (totalmente em Python e sem o uso da biblioteca de estrutura de dados Lemon), o tempo de processamento reduziu em pelo menos 3 vezes (a Tabela 5.1 compara o tempo necessário para calcular uma quantidade x de números primos com implementação em C e Python e execução da versão Python usando o interpretador de Python puro e o PyPy).

Por ser tão lento em um contexto que necessita de resultados rápidos, a utilização pura de Python foi descartada. A utilização do PyPy também foi descartada, pois a biblioteca Lemon é incompatível com o PyPy por ter dependência com a biblioteca SciPy de cálculos científicos que não foi portada para este interpretador. Estes argumentos foram suficientes para utilizar uma biblioteca compatível com ambas as linguagens Python e C++ (Lemon) para construir o gerente Nemo híbrido.

A implementação híbrida ficou restrita ao gerente Nemo. Como nos agentes não existe processamento pesado, sua implementação foi feita totalmente em Python, o que facilitou o seu rápido desenvolvimento. Por se tratar de uma linguagem interpretada, foi necessário criar um executável com as bibliotecas e o interpretador embutidos em um único binário usando o `py2exe`. Além disso, como a publicação dos agentes foi via GPO, criou-se um pacote MSI utilizando a biblioteca `pywin32` (Python Win32 *Extensions*).

O processo de construção e empacotamento do MSI utilizando o `py2exe` e o `pywin32`, depende de bibliotecas específicas e da versão do sistema operacional. Ou seja, para compatibilidade total é necessário que o pacote MSI seja construído no sistema operacional mais antigo. Como o ambiente de produção da UFMS utiliza apenas duas versões do sistema operacional Windows no domínio do AD, o Windows XP e o Windows 7, isso significa que para que o pacote MSI gerado seja compatível com ambos os sistemas, ele deve ser gerado no Windows XP.

Para a construção do GI foi utilizada a biblioteca de estruturas de dados Lemon,

Quantidade de Primos	C	Python	PyPy
10	0,001s	0,022s	0,039s
100	0,002s	0,033s	0,042s
1000	0,109s	2,463s	0,302s
5000	3,915s	1m30,847s	9,252s

Tabela 5.1: *Tempo médio de execução para calcular uma quantidade x de números primos.*

como mencionado no parágrafo anterior, compatível com ambas as linguagens. Durante o processo de construção do GI, cada agente envia seu GDS local para o gerente, que agrega essa informação em uma estrutura de dicionário, cuja chave é o IP na rede do host que hospeda o agente. Este formato de agregação evita problemas de colisão de informações específicas de cada GDS. A Figura 5.8 mostra como é a estrutura lógica do GI no gerente. Contudo, a facilidade de manipulação do GI utilizando o Lemon apresenta um custo em termos de espaço de memória e processamento. Um GI com 500mil nós e sem o corte da *d*-separação, ocupa cerca de 14GB em memória nos testes simulados. Este é um detalhe de implementação importante para uma ferramenta real de análise de anomalias.

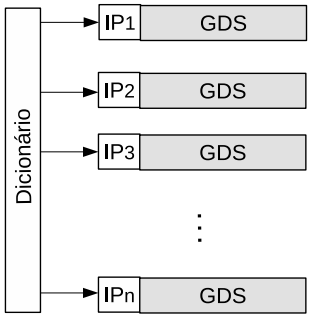


Figura 5.8: *Estrutura lógica do GI dentro do gerente.*

Com a estrutura construída em memória, o gerente pode executar os algoritmos de busca de anomalias. Essas execuções são realizadas quando requisitadas por um ou mais agentes na rede. Quando a requisição acontece, o gerente realiza uma cópia profunda do GI e não para de atualizar o GI original com novas informações. Para evitar que novas execuções de busca sejam realizadas, o gerente bloqueia e só aceita novas requisições de execução de busca após o término da primeira requisição. Uma modificação proposta, que não foi implementada, foi a possibilidade de enfileirar pedidos e versões do GI ao invés de descartar requisições até o término de uma execução. Como o tempo de execução do algoritmo de busca por anomalias pode demorar alguns segundos (Seção 6.4), algumas requisições de execução podem ser descartadas. Contudo, espera-se que os agentes estejam reportando problemas pelas mesmas causas, reduzindo a perda de informações.

O processo de requisição de execução dos algoritmos de busca de anomalias por um agente, além de outras funcionalidades, utiliza um protocolo simples de aplicação. Este protocolo possui três funcionalidades:

1. Enviar o GDS para o gerente;
2. Requisitar a execução do algoritmo de busca de anomalias;
3. Enviar informações de um serviço com seus tempos de acesso quando não preenchida a janela mínima de 200 requisições para um mesmo serviço em menos de 10 minutos (essa janela foi definida como parâmetro de configuração de Nemo).

Por ser um protocolo bem simples, pois um número de operação é enviado sempre no cabeçalho de uma operação, poderiam ter sido implementadas inúmeras outras funcionalidades (máximo de 65535 operações distintas). A cada nova funcionalidade implementada no agente, é necessário atualizar o pacote MSI com uma nova versão (modificando a entrada `version` no *script* de criação do MSI) superior à anterior, para que o AD consiga atualizar as versões nos clientes.

A utilização do Lemon facilita não só a manipulação do GI em memória mas também a impressão do grafo em um arquivo na linguagem `dot` (Graphviz). Quando o arquivo dot é compilado, ele gera um grafo que pode ser visualizado em `png` ou `pdf`. Os grafos de inferência, como o da Figura 5.3, foram gerados utilizando esta funcionalidade do Lemon com o Graphviz.

O gerente em conjunto com os agentes Nemo são capazes de detectar anomalias rapidamente e os resultados são tão precisos quanto os de Sherlock. A ferramenta funciona em modo texto no lado servidor e como processo no lado cliente. Faltou na implementação do gerente apenas uma ferramenta capaz de visualizar em tempo real partes do GI, e de executar algoritmos de detecção de anomalias diferentes. Foi criado um protótipo que deve ser evoluído para agregar mais funcionalidades e tornar a ferramenta final útil em ambientes de produção. A Figura 5.9 mostra o protótipo do visualizador de Nemo.

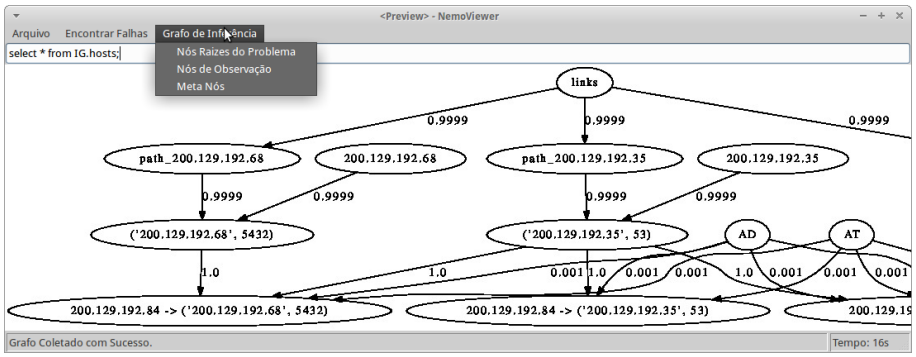


Figura 5.9: *Implementação inicial de um visualizador do GDS gerado por Nemo.*

Um ponto interessante da ferramenta, e que foi observado durante os testes do sistema, é que ela detectou um serviço desconhecido pela equipe de administração da rede. Um usuário havia instalado um Gerente SNMP em um Campus da instituição e não havia comunicado à equipe de suporte. Embora esse não seja o objetivo da ferramenta, ela

pode ser utilizada para detectar serviços que ferem a política de uso da rede como, por exemplo, comunidades P2P e serviços não homologados.

5.7 Considerações Finais

Neste capítulo foi apresentada a ferramenta Nemo. Ela é capaz de detectar anomalias em ambientes distribuídos usando como entrada GDSs agregados com informações da topologia da rede. O processo de detecção de uma anomalia na ferramenta Nemo depende do gerente central (que recebe os GDSs e a topologia de rede e constrói o GI) e, geralmente, é iniciado por uma requisição de um agente Nemo. Durante o processo de detecção, Nemo realiza operações de redução do GI original usando um conceito matemático sólido de Redes Bayesianas, a *d*-separação, e aplica uma função de pontuação diferente da sugerida por Sherlock. Os resultados de Nemo são uma lista contendo os possíveis nós da rede que fazem parte do problema. No próximo capítulo os resultados de Nemo serão comparados com os de Sherlock e de Spotlight.

Capítulo 6

Resultados Experimentais

Neste capítulo são apresentados resultados de simulação para avaliar vários aspectos de desempenho de Nemo. Embora Nemo tenha sido implementado e testado em um ambiente de produção, a avaliação por simulação permite que cenários maiores e mais complexos sejam explorados, além de se tratar de um ambiente controlado e passível de reprodução dos resultados.

O processo de simulação avalia inicialmente o quão eficiente é o algoritmo de redução do Grafo de Inferência (GI). A importância desse resultado está no fato de que, se o grafo resultante for muito menor, espera-se que ele necessite de menos espaço em memória e de menor tempo para detecção de anomalias. Além disso, esta abordagem pode ser útil em outras estratégias que utilizem GIs modelados por Redes Bayesianas na detecção de anomalias.

Em seguida, Nemo é comparado com seus antecessores Sherlock (2007) e Spotlight (2010). A modelagem das três soluções é semelhante, pois todas usam o GI como modelo. A principal diferença está nas manipulações no GI antes de detectar a anomalia e nas funções de pontuação das possíveis origens de uma anomalia.

Comparando-se Sherlock e Spotlight, o último é uma proposta mais recente e bem mais rápida que Sherlock. Entretanto, Spotlight utiliza uma heurística para reduzir o GI em um grafo bipartido que não preserva as propriedades teóricas do modelo inicial (Figura 4.22). Além disso, os resultados de precisão de Spotlight são bem inferiores aos de Sherlock.

Para fins de comparação, são utilizadas as seguintes métricas: precisão e tempo de execução. As próximas seções descrevem a metodologia de avaliação e apresentam os resultados da redução do GI e dados comparativos entre as três abordagens estudadas.

6.1 Metodologia de Avaliação

Para a avaliação, foram gerados aleatoriamente diversos GIs correspondentes a topologias de rede, dependências de serviços e distribuições de tempo de acesso. A metodologia

utilizada para geração do grafo é a mesma utilizada em [2, 20]. Os tempos de acesso são gerados de forma aleatória utilizando distribuições normais com médias (ativo e problemático) e desvios padrão obtidos a partir de dados reais coletados pelos agentes de Nemo na rede da UFMS. Foram coletados dados por duas semanas antes de se iniciar os experimentos simulados. As probabilidades de dependência são geradas aleatoriamente com distribuição uniforme.

As simulações foram realizadas em dois servidores, sendo um deles com dois processadores Intel Xeon E5530 x64, 32GB de RAM e outro com um processador Intel Xeon E5530 x64 com 16GB de RAM, ambos com sistema operacional Ubuntu Server Linux 12.04 LTS x64 com *kernel* 3.2.0. Todos os resultados foram obtidos a partir da média de pelo menos 50 execuções independentes de cada simulação. Além disso, para garantir a confiabilidade dos resultados, foi calculado um intervalo de confiança de 95% para cada resultado.

Para cada execução, o grafo gerado foi copiado em um grafo temporário antes de ser testado em cada abordagem. Dessa forma, todos os algoritmos foram testados com o mesmo grafo de entrada a cada passo da simulação. A quantidade de memória utilizada pelo grafo e o tempo de execução de cada simulação dependem diretamente da quantidade de nós do GI e da quantidade de anomalias injetadas. Um problema para o tempo de execução é a quantidade de anomalias, pois quanto mais anomalias, maior é a quantidade de combinações de vetores de atribuição. A quantidade de anomalias injetadas simultaneamente produz $(2 * r)^k$ vetores de atribuição, em que r é a quantidade de vetores e k a quantidade de anomalias. Por isso, quanto mais anomalias, maior o tempo de execução da simulação.

Para se avaliar a precisão, anomalias devem ser injetadas artificialmente nos grafos de inferência. Uma anomalia é introduzida selecionando-se aleatoriamente um nó raiz de problema e marcando-o como problemático. Após essa escolha, a anomalia deve ser propagada e refletida nos nós de observação. A propagação é feita por meio de um passeio probabilístico no grafo de inferência utilizando-se as probabilidades de dependência entre os nós (arestas). Observe que uma anomalia injetada em um nó raiz de problema pode refletir em vários nós de observação. A Figura 6.1 mostra que após dois nós raízes do problema ($k = 2$) serem marcados como problemáticos, o passeio probabilístico selecionou apenas alguns nós de observação. Estes nós são aqueles que reportam falhas para o início do processo de detecção dos prováveis causadores de uma anomalia e são chamados de sintomas.

Um algoritmo de inferência acerta quando indica corretamente todos os nós raízes de problema que foram marcados inicialmente como problemáticos. Esse modelo de análise de precisão é discutido em detalhes na Seção 6.3. Embora a taxa de acerto de 100% dos nós problemáticos seja o ideal, na prática, descobrir pelo menos um a cada execução do algoritmo pode auxiliar os administradores de rede na aceleração da busca por anomalias.

A geração dos grafos de inferência variou a quantidade de nós raízes de problema de 100 a 20.000 para avaliação do algoritmo de redução e de 150 a 250 para avaliação da precisão e tempo de execução dos algoritmos de inferência. A quantidade de anomalias injetadas variou de 1 a 4. A quantidade de nós raízes de problema variou mais na avaliação

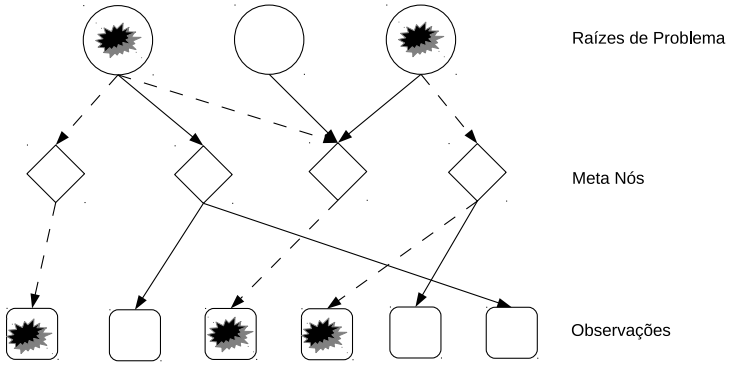


Figura 6.1: *Exemplo de GI com injeção de falhas nos nós raízes de problema e que após passeio probabilístico aleatório tem alguns nós de observação selecionados. No exemplo, dois nós raízes de problema são marcados como problemáticos e após o passeio probabilístico (linhas tracejadas) somente três dos seis possíveis nós de observação são marcados como nós que estão reportando problemas.*

da redução para demonstrar que o GI reduz significativamente quando existem muitos nós que podem ser a origem de uma anomalia.

6.2 Redução do Grafo de Inferência

O Algoritmo 2, apresentado na Seção 5.4, é executado para reduzir a quantidade de nós do GI com o objetivo de reduzir o consumo de memória do servidor e o tempo de execução do processo de diagnóstico de anomalias. Desta forma, a redução mantém somente os nós relacionados às anomalias no grafo, descartando os que não auxiliam na detecção do problema.

Neste experimento, os grafos utilizados na avaliação do algoritmo foram gerados variando as quantidades de anomalias injetadas e de nós raízes de problema. Para cada combinação, foram gerados 100 grafos diferentes utilizando-se a metodologia descrita na Seção 6.1. Pode-se observar na Figura 6.2 que a taxa de redução apresentada é diretamente proporcional a quantidade de nós raízes de problema e inversamente proporcional a quantidade de anomalias. A taxa de redução variou entre 50% até pouco mais de 99%. Essa redução elevada é explicada pela alta quantidade de nós raízes de problema que não explicam os sintomas provocados pelas anomalias, como, por exemplo, roteadores ou enlaces que não estão no caminho da requisição do cliente. A figura também mostra os intervalos de confiança de 95% para a média.

Embora a redução seja significativa, ela será útil somente se o processo de diagnóstico não for afetado. Isso significa que o processo de redução deve ser seguro o suficiente para não remover nós importantes ao processo de detecção de anomalias. Além disso, a redução também possui um custo computacional que, embora pequeno, pode piorar o tempo de execução de algumas abordagens. No caso do Spotlight, a redução do tamanho do GI não acelerou o processo e nem melhorou significativamente a taxa de acertos. Por isso, usar o processo de redução com o Spotlight não se mostrou vantajoso. Assim, dependendo

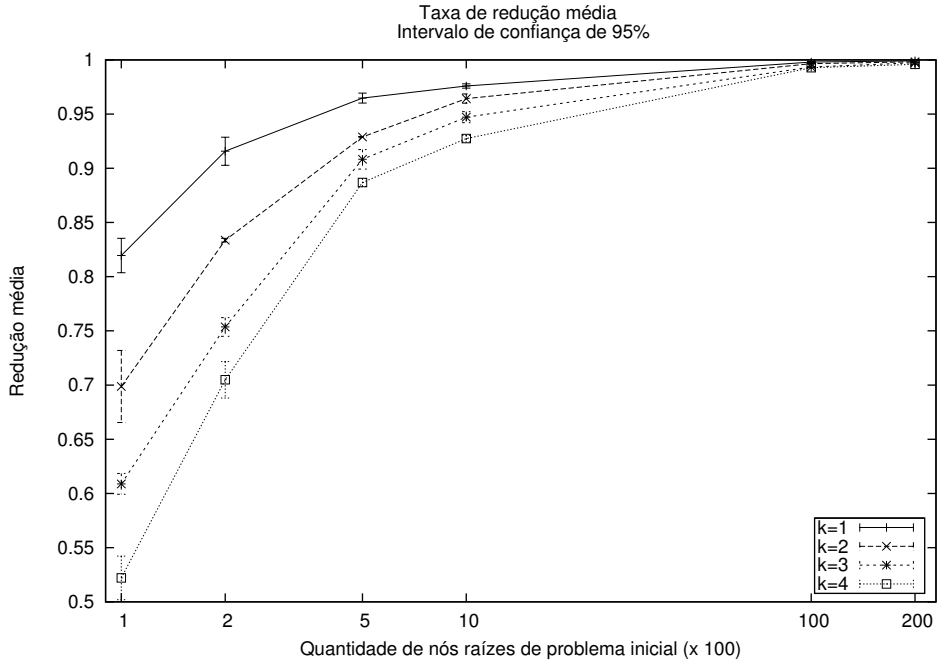


Figura 6.2: *Taxa de redução média de nós raízes de problema, com um intervalo de confiança de* 95%*, de grafos com diferentes quantidades (*100*,* 200*,* 500*,* 1k*,* 10k *e* 20k*) de nós raízes de problema e com* 1 *até* 4 *anomalias simultâneas.*

da forma como o GI é manipulado, o custo computacional da redução pode até piorar o tempo de execução total. Contudo, em uma ferramenta de produção real, quanto menor a quantidade de memória utilizada pelo gerente, melhor. Por isso, a redução pode até aumentar o tempo de execução de um algoritmo de inferência, mas o uso de memória no gerente com certeza será reduzido.

6.3 Precisão

A precisão de uma abordagem é definida como a razão entre a quantidade de execuções com detecções corretas pela quantidade total de execuções. A Figura 6.3 mostra a precisão das abordagens com GIs de quantidade fixa de nós de observação, diferentes quantidades de nós raízes de problema e anomalias simultâneas. Nas comparações, Nemo se mostrou superior ao Sherlock [2] e ao Spotlight [20] em todos os cenários estudados. Os resultados mostram que, embora mais rápido, o Spotlight é menos preciso que as duas outras abordagens.

Embora os valores de precisão para duas e três anomalias sejam baixos, os resultados apresentados consideram como acerto somente quando todas as anomalias são corretamente selecionadas e estão no vetor de atribuição com melhor pontuação. Entretanto, Nemo identifica corretamente pelo menos uma das anomalias em mais de 82% dos casos, conforme mostra a Tabela 6.1. Quando se considera os cinco melhores vetores de atribuição, Nemo acerta pelo menos uma das anomalias em mais de 97% dos casos.

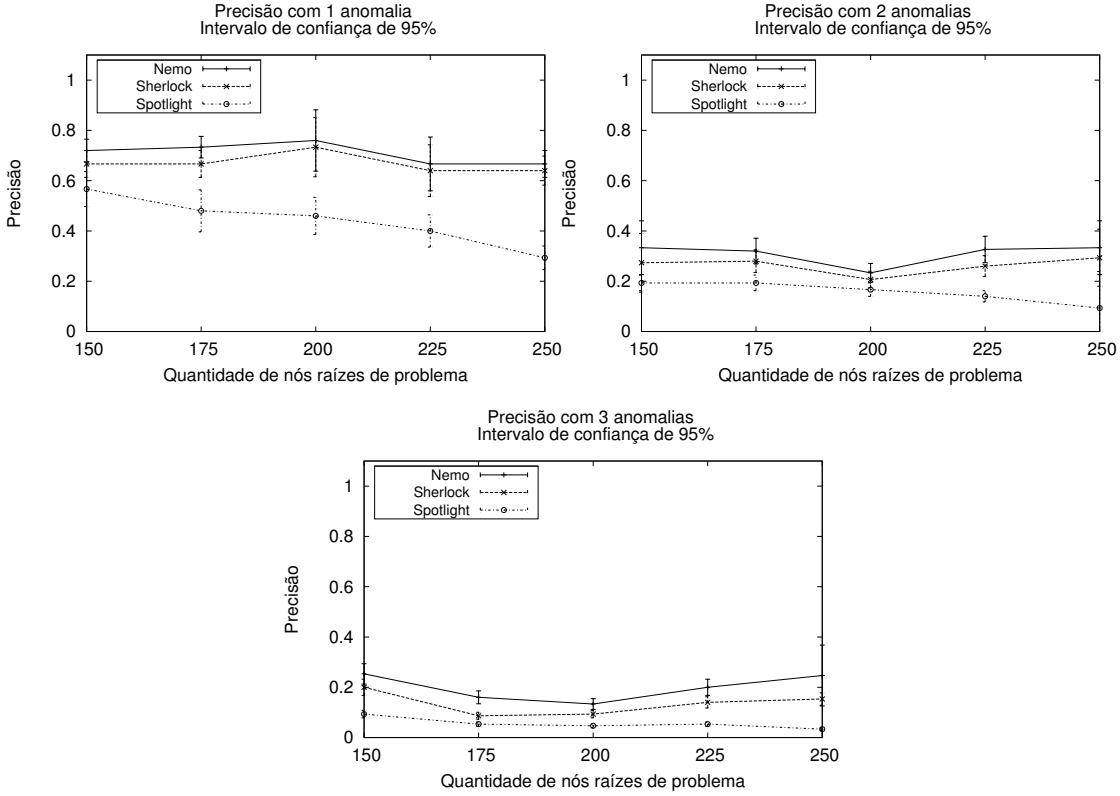


Figura 6.3: *Comparação da precisão das abordagens com uma, duas e três anomalias simultâneas, variando a quantidade de nós raízes de problema.*

Tabela 6.1: *Taxa de detecção correta de pelo menos uma anomalia.*

Quantidade de nós raízes de problema	Taxa de detecção correta					
	2 anomalias			3 anomalias		
	Nemo	Sher.	Spot.	Nemo	Sher.	Spot.
150	82,0	79,3	73,3	97,3	95,3	91,3
175	87,3	84,6	64,6	94,6	91,3	72,6
200	91,3	87,3	65,3	95,3	92,0	78,6
225	92,0	90,6	62,0	92,6	91,3	67,3
250	91,3	90,0	58,0	94,6	93,3	60,0

Em uma rede corporativa existem muitos nós raízes de problema. Quando um problema é reportado, é difícil detectar instantaneamente todas as possíveis origens de uma anomalia. Por isso, se for possível para os administradores de rede extrair de uma ferramenta pelo menos um problema por vez, será possível direcionar os próximos diagnósticos antes de uma solução final. Nemo acerta em 97% dos casos uma ou mais anomalias nos cinco melhores conjuntos de hipóteses retornados. Desta forma, a verificação de hipóteses com cinco diferentes pontuações é plausível em um processo de diagnóstico de anomalias

em redes corporativas, pois a ferramenta extrai pelo menos um problema em cada iteração. Assim, o processo corretivo pode ser desenvolvido de forma iterativa, corrigindo-se inicialmente as anomalias que foram indicadas corretamente e repetindo-se o processo até que todas sejam resolvidas. Além disso, os resultados dos cinco melhores vetores de atribuição (97% de acerto) poderiam ser combinados, em um trabalho futuro, com o resultado em que Nemo retorna precisamente as anomalias (aproximadamente 26%, com três anomalias e 150 raízes de problema), melhorando a precisão sem sacrificar o tempo de execução.

Tabela 6.2: *Taxa de acertos do Sherlock com a função de pontuação original, função de pontuação do Nemo, com e sem a* d*-separação. (PD = Sherlock com função de pontuação original, DI = Sherlock com função de pontuação discretizada, DP = Sherlock após* d*-separação) com função de pontuação original, DD = Sherlock após* d*-separação com função de pontuação discretizada).*

Quantidade de nós raízes de problema	Taxa de acertos do Ferret e modificações							
	1 anomalia				2 anomalias			
	PD	DI	DP	DD	PD	DI	DP	DD
150	69,0	78,0	71,0	79,0	25,0	32,0	29,0	35,0
175	64,0	74,0	67,0	75,0	30,0	37,0	33,0	37,0
200	73,0	79,0	73,0	79,0	19,0	27,0	21,0	28,0
225	67,0	72,0	67,0	72,0	24,0	29,0	25,0	28,0
250	63,0	72,0	65,0	72,0	28,0	33,0	30,0	35,0

A Tabela 6.2 compara a precisão do Sherlock com as modificações de Nemo. Isso foi feito para demonstrar que a utilização da *d*-separação e da nova função da pontuação melhoram os resultados da abordagem original. Dessa vez foram realizadas 100 simulações para cada conjunto de raízes de problema e anomalias. A tabela mostra que a função de pontuação de Nemo melhora a precisão muito mais do que a *d*-separação. Contudo, a *d*-separação também aumentou a precisão em alguns casos. Isso ocorre, pois somente os nós do GI que estão relacionados de fato com o problema são analisados. Por isso, Nemo é mais eficiente que Sherlock, pois a função de pontuação tem melhores resultados e a *d*-separação não reduz a precisão.

6.4 Tempo de execução

A estratégia global de Nemo é reduzir o tamanho do grafo para melhorar o tempo de execução dos algoritmos de inferência. Esperava-se que ao reduzir o tamanho do GI, o tempo de execução reduzisse também, uma vez que a quantidade de nós raízes de problema teve redução superior a 50% (no pior caso), como demonstrado na Seção 6.2. Como um dos gargalos computacionais do algoritmo de inferência é a geração de todos os possíveis vetores de atribuição, isso significa, que quanto menor a quantidade de nós raízes de problema, mais rápido deve ser o algoritmo de inferência. Além disso, o processo de propagação de estados também deve ser reduzido, pois o GI, após o processo de redução, tem uma quantidade menor de nós.

Quando Nemo e Sherlock são comparados, Nemo supera Sherlock em tempo de execução em muitas vezes. E não só isso. A precisão de Nemo também é superior. Por se tratar de um processo automatizado, de forma empírica, é fácil concluir que quanto menor a quantidade de possíveis raízes de problema, maior a possibilidade de uma ferramenta acertar corretamente as origens de problemas. Entretanto, Spotlight é mais rápido que Nemo quando duas e três anomalias são injetadas. Como Spotlight tem um modelo próprio de redução do GI e um algoritmo de cobertura que não explora todas as possíveis possibilidades, ele é mais rápido que Nemo e Sherlock, porém, é menos preciso que ambos. Além disso, Spotlight, ao contrário de Nemo e Sherlock, é incapaz de produzir múltiplas hipóteses. O algoritmo de redução também foi testado com Spotlight, mas não resultou em redução significativa dos tempos de execução e, em alguns casos, Spotlight com redução teve tempo superior e sem grandes ganhos em sua precisão em relação a Spotlight original.

Tabela 6.3: *Tempo médio de execução das três abordagens.*

Quantidade de nós raízes de problema	Tempo médio de execução (s)								
	1 anomalia			2 anomalias			3 anomalias		
	Nemo	Sher.	Spot.	Nemo	Sher.	Spot.	Nemo	Sher.	Spot.
150	0,0028	0,0118	0,0064	0,0858	2,5260	0,0026	6,2356	653,4460	0,0228
175	0,0018	0,0180	0,0016	0,1162	3,9704	0,0034	6,7199	1548,27	0,0560
200	0,0032	0,0196	0,0004	0,1188	4,7354	0,0040	7,4836	3853,56	0,0584
225	0,0038	0,0198	0,0054	0,1104	6,1285	0,0046	24,1810	8289,36	0,0570
250	0,0056	0,0236	0,0050	0,1210	7,8026	0,0062	32,4695	14960,9	0,0672

A Tabela 6.3 compara os tempos de execução das três abordagens quando executadas com os grafos de inferência utilizados na Seção 6.2. Como esperado, Nemo é bem mais rápido que Sherlock em todos os casos, mas não tão rápido quanto Spotlight. Com apenas uma anomalia Nemo consegue ser mais rápido que Spotlight em alguns casos, contudo, conforme a quantidade de anomalias aumenta, Nemo começa a perder. Já o Sherlock demora tanto que se torna uma solução inviável na prática.

Tabela 6.4: *Tempo médio do Sherlock com intervalo de confiança de* 95%.

Qte nós raízes de problema	Tempo médio de execução (s) do Sherlock								
	1 anomalia			2 anomalias			3 anomalias		
	$\mu - \epsilon$	μ	$\mu + \epsilon$	$\mu - \epsilon$	μ	$\mu + \epsilon$	$\mu - \epsilon$	μ	$\mu + \epsilon$
150	0,0112	0,0118	0,0123	2,5215	2,5260	2,5304	651,43	653,45	655,46
175	0,0174	0,0180	0,0185	3,9058	3,9704	4,0349	1543,68	1548,28	1552,88
200	0,0194	0,0196	0,0197	4,6837	4,7354	4,7870	3852,81	3853,57	3854,32
225	0,0197	0,0198	0,0054	6,1233	6,1285	6,1238	8267,92	8289,36	8310,81
250	0,0225	0,0236	0,0246	7,7458	7,8026	7,8593	14639,56	14960,93	15282,30

Os tempos médios observados na Tabela 6.3 foram extraídos da Tabela 6.5, Tabela 6.4 e Tabela 6.6. Essas tabelas apresentam os resultados das simulações com intervalo de confiança de 95%. Além disso, para cada resultado, foram executadas 50 rodadas independentes de simulação. Um detalhe importante é que, pelo fato do intervalo de confiança

Tabela 6.5: *Tempo médio do Spotlight com intervalo de confiança de* 95%.

Qte nós raízes de problema	Tempo médio de execução (s) do Spotlight								
	1 anomalia			2 anomalias			3 anomalias		
	$\mu - \epsilon$	μ	$\mu + \epsilon$	$\mu - \epsilon$	μ	$\mu + \epsilon$	$\mu - \epsilon$	μ	$\mu + \epsilon$
150	0,0053	0,0064	0,0074	0,0018	0,0026	0,0033	0,0000	0,0228	0,0612
175	0,0011	0,0016	0,0020	0,0024	0,0034	0,0043	0,0431	0,0560	0,0688
200	0,0002	0,0004	0,0005	0,0028	0,0040	0,0051	0,0420	0,0584	0,0747
225	0,0041	0,0054	0,0066	0,0033	0,0046	0,0058	0,0410	0,0570	0,0729
250	0,0036	0,0050	0,0064	0,0051	0,0062	0,0072	0,0356	0,0672	0,0987

Tabela 6.6: *Tempo médio do Nemo com intervalo de confiança de* 95%.

Qte nós raízes de problema	Tempo médio de execução (s) do Nemo								
	1 anomalia			2 anomalias			3 anomalias		
	$\mu - \epsilon$	μ	$\mu + \epsilon$	$\mu - \epsilon$	μ	$\mu + \epsilon$	$\mu - \epsilon$	μ	$\mu + \epsilon$
150	0,0020	0,0028	0,0035	0,0701	0,0858	0,1014	5,0300	6,2356	7,4411
175	0,0012	0,0018	0,0023	0,0976	0,1162	0,1347	5,3087	6,7199	8,1311
200	0,0023	0,0032	0,0040	0,0911	0,1188	0,1464	6,9834	7,4836	7,9837
225	0,0027	0,0038	0,0048	0,0850	0,1104	0,1357	16,9516	24,1810	31,4103
250	0,0040	0,0056	0,0071	0,0871	0,1210	0,1548	32,3854	32,4695	32,5537

das abordagens representar medidas de tempo, isso significa que não pode existir um valor negativo. Na Tabela 6.5, com 150 nós raízes de problema, um dos campos ficou zerado, pois o seu extremo inferior teve resultado negativo.

6.5 Considerações Finais

Neste capítulo foram realizados três testes em ambiente controlado com a ferramenta Nemo. O primeiro teste avaliou a taxa média de redução de um GI quando aplicado o algoritmo de *d*-separação. O segundo teste avaliou a precisão de Nemo em relação a Sherlock e Spotlight quando injetadas até 4 anomalias simultâneas em um GI. O terceiro e último teste comparou o tempo de execução de Nemo em relação a Sherlock e Spotlight. Outra comparação importante deste capítulo foi a apresentada na Tabela 6.2, que compara as modificações de Nemo realizadas no algoritmo Ferret e mostra que a *d*-separação aliada à função de pontuação de Nemo aumentam a precisão na detecção de anomalias.

Nemo é mais preciso que Sherlock e Spotlight, mas inferior a Spotlight em tempo de execução. Por outro lado, Spotlight é mais impreciso conforme a quantidade de anomalias injetadas simultaneamente aumenta. Por isso, Nemo é uma boa solução para detecção de anomalias, pois não perde informações importantes do GI e detecta anomalias com maior precisão em relação aos trabalhos anteriores.

Capítulo 7

Conclusão

O estudo de detecção de anomalias em ambientes distribuídos está longe de ser exaurido. Novas técnicas [43] exploram problemas de desempenho e falhas em *data centers* ou novas formas de mapear e correlacionar dependências [44, 30] de rede, além da construção de novas ferramentas para auxiliar os administradores de rede.

Neste trabalho foram avaliadas diversas ferramentas e propostas sobre detecção de anomalias em ambientes distribuídos. Essa área não está limitada aos exemplos intrusivos e não intrusivos citados. Ao longo do trabalho, várias outras propostas foram estudadas [6, 8, 26, 38, 27, 37] e serviram de base para aprimorar o conhecimento e conhecer as diferentes técnicas de modelagem do problema (Redes Bayesianas, análise de sinais etc.).

Nemo foi projetado com o objetivo de se tornar uma ferramenta real e automatizada de detecção de anomalias em uma rede corporativa. Com esse objetivo definido, alguns aspectos precisaram ser delimitados. Uma aplicação real deve ser facilmente instalada na rede. Por isso, os agentes Nemo foram desenvolvidos para publicação automatizada via *Active Directory*. Não limitado a isso, Nemo também nasceu multiplataforma e por isso funciona também em ambientes Unix-*like*, como o Linux. Além disso, as linguagens e bibliotecas foram selecionadas para permitir futuras expansões da ferramenta, como uma interface de gerência ou formas de tratamentos e relatórios de incidentes.

Para a construção da ferramenta Nemo, dois trabalhos foram base de comparação: Sherlock [2] e um trabalho mais recente derivado deste, o Spotlight [20]. Por isso, a ferramenta também é não intrusiva e não modifica aplicações, servidores ou sistemas legados. Ela utiliza somente traços de rede para inferir as possíveis causas dos problemas, sendo particularmente útil para ambientes legados. Embora Nemo não possua um módulo específico de captura de pacotes em equipamentos roteadores, sua construção é possível, pois basta que esteja bem definida a origem e destino dos pacotes nos traços de rede para que Nemo detecte possíveis anomalias.

Este trabalho contribui com a implementação de uma ferramenta de detecção de anomalias, Nemo, um algoritmo eficiente para redução de GIs que sejam modelados como Redes Bayesianas e uma nova função de pontuação para inferência de anomalias que explora conhecimento específico do domínio do problema. A ferramenta, embora não

possua uma interface funcional, é capaz de monitorar uma rede corporativa detectando possíveis anomalias, além da possibilidade de instalação do seu agente nos mais diversos sistemas operacionais, dada a natureza multiplataforma da linguagem de programação escolhida. O algoritmo de redução mostrou-se capaz de reduzir um GI modelado em uma rede bayesiana em mais de 50% e, em alguns casos, mais de 99% do tamanho do grafo original. E, finalmente, a função de pontuação, no processo de discretização do problema, melhorou a precisão do diagnóstico em todos os cenários avaliados.

O código fonte da ferramenta está disponível no endereço `http://ndsg.facom.ufms.br/nemo`. Além disso, os principais resultados obtidos neste trabalho foram submetidos ao SBRC2013 [21].

7.1 Trabalhos Futuros

Nemo é uma ferramenta promissora, mas ainda incompleta. Nemo possui um protótipo de ferramenta de visualização que está inacabado e é pouco funcional (Figura 5.9). Além disso, a implementação de persistência é parcial e utiliza um banco de dados relacional SQL (`sqlite`), que pode ser um gargalo dependendo da quantidade de requisições e modificações no GI. Possíveis melhorias seriam utilizar um banco de dados mais robusto ou um modelo de armazenamento em disco do GI.

Nemo não possui qualquer módulo capaz de armazenar um histórico de problemas de um nó da rede e nem gerar relatórios de eventos ou alertas de sobrecarga. Para Nemo se tornar uma ferramenta produtiva, além de detectar anomalias, deve ser capaz de mostrar trechos do GI, executar algoritmos de inferência via interface e possuir os módulos citados acima. Uma outra modificação que pode melhorar ainda mais Nemo é a possibilidade de cruzar informações de mais de um método de inferência como, por exemplo, comparar os resultados de Nemo e Spotlight e mostrar para o administrador de rede apenas os resultados comuns às duas abordagens. Esse cruzamento é factível, pois Nemo e Spotlight são muito rápidos quando comparados a Sherlock.

Referências Bibliográficas

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, e A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. Em *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, páginas 74–89. New York, NY, USA, 2003.

[2] P. Bahl, R. Chandra, A. G. Greenberg, S. Kandula, D. A. Maltz, e M. Zhang. Towards Highly Reliable Enterprise Network Services Via Inference of Multi-Level Dependencies. Em *Proceedings of the 2007 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*, páginas 13–24. New York, NY, USA, 2007.

[3] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, e A. Simma. Constellation: Automated Discovery of Service and Host Dependencies in Networked Systems. *Technical Report, MSR-TR-2008-67, Microsoft Research*, 2008.

[4] P. Barham, A. Donnelly, R. Isaacs, e R. Mortier. Using Magpie for Request Extraction and Workload Modelling. Em *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, páginas 259–272. Berkeley, CA, USA, 2004.

[5] C. M. Bishop. Pattern Recognition and Machine Learning. Springer, 2006.

[6] A. Chanda, A. L. Cox, e W. Zwaenepoel. Whodunit: Transactional Profiling for Multi-tier Applications. Em *Proceedings of the 2nd EuroSys European Conference on Computer System (Eurosys'07)*, páginas 17–30. 2007.

[7] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, e E. A. Brewer. Path-Based Failure and Evolution Management. Em *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, páginas 309–322. Berkeley, CA, USA, 2004.

[8] X. Chen, M. Zhang, Z. M. Mao, e P. Bahl. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. Em *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*, páginas 117–130. Berkeley, CA, USA, 2008.

[9] Cisco. Netflow Network Protocol. `http://www.cisco.com/go/netflow`, 2009. [Online; acessado 12-Jan-2013].

[10] CoralCDN. The Coral Content Distribution Network. `http://www.coralcdn.org/`, 2004. [Online; acessado 12-Jan-2013].

[11] Transaction Processing Performance Council. Web-based ordering. `http://www.tpc.org/wspec.html`. [Online; acessado 12-Jan-2013].

[12] R. Fonseca, M. J. Freedman, e G. Porter. Experiences with Tracing Causality in Networked Services. Em *Proceedings of the Internet Network Management Conference on Research on Enterprise Networking (INM/WREN'10)*. Berkeley, CA, USA, 2010.

[13] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, e I. Stoica. X-Trace: A Pervasive Network Tracing Framework. Em *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI'07)*. Berkeley, CA, USA, 2007.

[14] Ethan Galstad. Nagios. `http://www.nagios.org/`, 1999. [Online; acessado 12-Jan-2013].

[15] Dan Geiger, Thomas Verma, e Judea Pearl. d-Separation: From Theorems to Algorithms. Em *UAI*, páginas 139–148. 1989.

[16] HP. HP Openview. `http://www.hp.com/united-states/outputmanagement/output_manager_openview.html`, 2011. [Online; acessado 12-Jan-2013].

[17] Henrik Härkönen. Python Optimization: Extending with C. `http://kortis.to/radix/python_ext/`, 2002. [Online; acessado 12-Jan-2013].

[18] IBM. IBM Tivoli. `http://www-01.ibm.com/software/br/tivoli/`, 2011. [Online; acessado 12-Jan-2013].

[19] V. Jacobson, C. P. Wood, T. Seaver, e K. Adelman. Traceroute - Print the Route Packets take to Network Host. `http://beej.us/guide/bgnet/`, 2008. [Online; acessado 12-Jan-2013].

[20] D. John, P. Prakash, R. R. Kompella, e R. Chandra. Shedding Light on Enterprise Network Failures Using Spotlight. Em *Proceedings of 29th IEEE Symposium on Reliable Distributed Systems (SRDS'10)*, páginas 167–176. 2010.

[21] B. A. Silva Jr, F. B. de Carvalho, e R. A. Ferreira. Nemo: Procurando e Encontrando Anomalias em Aplicações Distribuídas. Em *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'13) (submetido)*. Brasilia, DF, Brasil, 2013.

[22] S. Kandula, R. Chandra, e D. Katabi. What's Going On?: Learning Communication Rules in Edge Networks. Em *Proceedings of the 2008 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'08)*, páginas 87–98. New York, NY, USA, 2008.

[23] S. Kandula, D. Katabi, e J. P. Vasseur. Shrink: A Tool For Failure Diagnosis in IP Networks. Em *Proceedings of MineNet Workshop at 2006 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'06)*, páginas 173–178. 2005.

[24] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, e P. Bahl. Detailed Diagnosis in Enterprise Networks. Em *Proceedings of the 2009 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'09)*, páginas 243–254. New York, NY, USA, 2009.

[25] R. R. Kompella, J. Yates, A. G. Greenberg, e A. C. Snoeren. IP Fault Localization Via Risk Modeling. Em *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'05)*. 2005.

[26] E. Koskinen e J. Jannotti. BorderPatrol: Isolating Events for Black-box Tracing. Em *Proceedings of the 3rd EuroSys European Conference on Computer Systems (Eurosys'08)*, páginas 191–203. New York, NY, USA, 2008.

[27] P. Lucian, C. Byung-Gon, S. Ion, C. Jaideep, e T. Nina. Macroscope: End-Point Approach to Networked Application Dependency Discovery. Em *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'09)*, páginas 229–240. Rome, Italy, 2009.

[28] Gordon Lyon. Nmap Security Scanner. `http://nmap.org/`, 1997. [Online; acessado 12-Jan-2013].

[29] Microsoft MOM. Microsoft Operations Manager. `http://msdn.microsoft.com/en-us/library/aa505337.aspx`, 2012. [Online; acessado 12-Jan-2013].

[30] A. Natarajan, P. Ning, Y. Liu, S. Jajodia, e S. E. Hutchinson. NSDMiner: Automated discovery of Network Service Dependencies. Em *Proceedings of the IEEE (INFOCOM'12)*, páginas 2507–2515. 2012.

[31] Oasis. OASIS anycast service. `http://oasis.coralcdn.org/`, 2005. [Online; acessado 12-Jan-2013].

[32] T. Oetikers e D. Rand. The Multi Router Traffic Grapher. `http://oss.oetiker.ch/mrtg/index.en.html`, 2011. [Online; acessado 12-Jan-2013].

[33] P. Reynolds, C. Edwin K., J. L. Wiener, J. C. Mogul, M. A. Shah, e A. Vahdat. PIP: Detecting the Unexpected in Distributed Systems. Em *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*. Berkeley, CA, USA, 2006.

[34] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, e A. Vahdat. WAP5: Black-Box Performance Debugging for Wide-Area Systems. Em *Proceedings of the 15th international conference on World Wide Web (WWW'06)*, páginas 347–356. New York, NY, USA, 2006.

[35] A. Rigo, M. Fijatkowski, C. F. Bolz, A. Cuni, B. Peterson, H. Ardö, e H. Krekel. PyPy is a fast, compliant alternative implementation of the Python language. `http://pypy.org/`, 2002. [Online; acessado 12-Jan-2013].

[36] Michel Santos. JMeasure: An Object-Oriented Manner of Representing Measurements, Such as Lengh, Mass, Time, Currency. `http://jmeasure.sourceforge.net/`, 2003. [Online; acessado 12-Jan-2013].

[37] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, e C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. *Techinical Report 36356, Google Inc.*, 2010.

[38] V. K. Singh, H. Schulzrinne, e K. Miao. DYSWIS: An Architecture for Automated Diagnosis of Networks. Em *Network Operations and Management Symposium: Pervasive Management for Ubioquitous Networks and Services (NOMS'08)*, páginas 851–854. 2008.

[39] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, e R. N. Chang. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. Em *Proceedings of The 34th USENIX Annual Technical Conference (USENIX'09)*. Berkeley, CA, USA, 2009.

[40] Andrew M. Rudoff W. Richard Stevens, Bill Fenner. *Programação de Rede Unix, API para soquetes de rede*, volume 1. ARTMED Editora S.A, terceira edição, 2004.

[41] M. Welsh, D. Culler, e E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. Em *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'01)*, páginas 230–243. New York, NY, USA, 2001.

[42] G. Wilkins e Eclipse Foundation. The Jetty Web Server. `http://www.eclipse.org/jetty`, 2009. [Online; acessado 12-Jan-2013].

[43] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, e C. Kim. Profiling Network Performance for Multi-Tier Data Center Applications. Em *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'11)*. 2011.

[44] Y. Zhao, Y. Cao, Y. Chen, M. Zhang, e A. Goyal. Rake: Semantics Assisted Network-Based Tracing Framework. Em *Proceedings of 19th Internation Workshop on Quality of Service (IWQoS'11)*, páginas 1–9. San Jose, CA, USA, 2011.

Apêndice A

Implementação da *d*-separação em Python

O Algoritmo 1 da *d*-separação, descrito na Seção 5.4, recebe como entrada um grafo G e dois conjuntos de nós X e Z. O que este algoritmo faz na prática, é retornar o conjunto de nós R, ou no caso da implementação abaixo, `result`, que representa os nós que estão *d*-conectados com X em G por Z. A *d*-separação em si é o conjunto complementar da *d*-conexão. Isso significa que o conjunto de nós que estão *d*-separados são os nós da diferença de $G - R$. Abaixo segue a implementação da *d*-conexão utilizada como base para o resultado da *d*-separação.

```
# -*- coding: utf-8 -*-
import networkx as nx
import datetime

#Retorna os nos de um determinado tipo usando NetworkX
def return_nodes(G, ig_type='type', node_type=None):
    nodes = nx.get_node_attributes(G, ig_type)
    node_list = []

    if(node_type == None):
        return nx.nodes(G)

    for i in nodes.keys():
        if(nodes[i] == node_type):
            node_list.append(i)

    return node_list

#Retorna os ancestrais de um no
def ancestors(G, S):
    L = set()
    L.update(S)
```

```
    ancestors = set()

    while L:
        y = L.pop()
        if y not in ancestors:
            for k in G.in_edges(y):
                L.add(k[0])
        if not G.in_edges(y):
            ancestors.add(y)

    return ancestors

#Algoritmo utilizado para d-separacao
def d-conection(G, X, Z):
    class Direction: up, down = range(2)

    # first collect all ancestors of givens
    ancestors = set()         # ancestrais
    L = set()
    L.update(Z)               # nos que precisao ser visitados

    while L:
        y = L.pop()
        if y not in ancestors:
            for k in G.in_edges(y):
                L.add(k[0]    # precisa visitar os pais tambem
        ancestors.add(y)      # y e o acestral de uma evidencia

    # visitar as trilhas ativas comecando por x
    L = set()                 # (no, direcao) a serem visitados
    for i in X:
        L.add( (i, Direction.up) )
    visited = set()           # (no, direcao) previamente visitados
    result  = set()           # nos alcancaveis por uma trilha ativa

    while L:
        yd = L.pop()
        if yd in visited:
            continue
        visited.add(yd)
        y, d = yd
        y_not_given = y not in Z
        if y_not_given:
            result.add(y)     # y e alcancavel

        if d == Direction.up and y_not_given:
```

```
            children = []
            parents = []

            for k in G.out_edges(y):
                children.append(k[1])

            for k in G.in_edges(y):
                parents.append(k[0])

            L.update( (z, Direction.down) for z in children )
            L.update( (z, Direction.up)   for z in parents )

        elif d == Direction.down:
            if y_not_given:
                children = []

                for k in G.out_edges(y):
                    children.append(k[1])

                L.update( (z, Direction.down) for z in children )

            if y in ancestors:
                parents = []

                for k in G.in_edges(y):
                    parents.append(k[0])
                L.update( (z, Direction.up)   for z in parents )

    return result            #retorna o conjunto de nos R
```