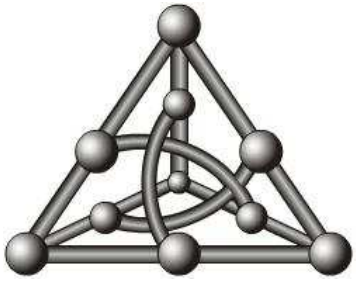
Construção de Caminhos Causais em Redes Definidas por Software

Fabrício Barbosa de Carvalho

Dissertação de Mestrado

Orientação: Prof. Ronaldo Alves Ferreira

Área de Concentração: Redes de Computadores



Faculdade de Computação

Universidade Federal de Mato Grosso do Sul

Outubro de 2014

Resumo

As redes corporativas atuais são compostas de diversos equipamentos e aplicações. O aumento da escala de uma rede corporativa faz com que as interações entre as aplicações se tornem mais complexas e envolvam diversos elementos de hardware e software, como: enlaces, *switches*, roteadores, servidores e sistemas operacionais. Consequentemente, determinar corretamente quais elementos foram utilizados no processamento de uma requisição nesse ambiente se torna uma tarefa extremamente desafiadora.

Diferentes técnicas de como determinar os elementos de hardware e software utilizados no processamento das requisições foram propostas na literatura. Esses elementos são agrupados em um conjunto denominado caminho causal. As técnicas de construção de caminhos causais são incorporadas em ferramentas de detecção de anomalias. Essas ferramentas detectam falhas ou sobrecargas nos elementos dos caminhos causais. Além disso, essas ferramentas são divididas em dois grandes grupos: as intrusivas e as não intrusivas. A principal diferença entre os grupos é que nas intrusivas as aplicações precisam ser modificadas para se inserir informações de controle, enquanto nas não intrusivas não há essa necessidade. O objetivo comum, entretanto, é mapear o caminho causal de uma requisição.

Todas as ferramentas de construção de caminhos causais e, consequentemente, para detecção de anomalias até então conhecidas foram concebidas para a arquitetura atual da Internet. Essa arquitetura tem sido fortemente criticada por ser de difícil alteração e evolução, sendo inclusive rotulada de ossificada por alguns pesquisadores. O novo paradigma de *Rede Definida por Software* (SDN - *Software Defined Network*) foi proposto para contornar os problemas da arquitetura atual da Internet. SDN

proporciona a dissociação entre o plano de controle e o plano de dados, permitindo que elementos externos de software exerçam funções do plano de controle e alterem o comportamento do plano de dados.

Este trabalho propõe S-Trace, uma ferramenta não intrusiva de construção de caminhos causais que explora aspectos positivos das ferramentas de construção de caminhos causais e de detecção de anomalias intrusivas e não intrusivas, além da separação de planos fornecida por SDN. S-Trace intercepta chamadas de função de bibliotecas para correlacionar os eventos de comunicação inter-processo (IPC - *Inter-Process Communication*) utilizados ao processar uma requisição. Além disso, S-Trace explora a separação de planos de SDN para realizar a reprodução de tráfego de rede e, assim, aprimorar os caminhos causais construídos.

Os resultados da avaliação mostram que S-Trace constrói caminhos causais corretamente, capturando o comportamento das aplicações ao processar as requisições. A avaliação utilizou duas aplicações que abrangem grande parte dos diferentes modelos de implementação utilizados pelas aplicações distribuídas de rede. Durante a avaliação, S-Trace foi capaz de construir caminhos causais mesmo na presença de falhas e de dispositivos NAT que modificam os endereços IP das conexões das requisições, fatores que dificultam significativamente a construção dos caminhos.

Palavras-chave: *caminhos causais, redes definidas por software, SDN, aplicações distribuídas.*

Abstract

Today's enterprise networks are composed of several devices and applications. The increase in scale of an enterprise network makes the interactions between applications more complex because they may include a significantly large number of hardware and software elements, such as links, switches, routers, servers and operating systems. Consequently, correctly determining the elements used in processing a request in this environment becomes an extremely challenging task.

Several techniques to determine the hardware and software elements used to process a request have been proposed in the literature. These elements are grouped into a set called causal path. Techniques for building causal paths are incorporated in anomaly detection tools in order to detect faults and overloads on elements of the causal paths. These tools are divided into two major groups: intrusive and non-intrusive. The main difference between the groups is that the intrusive tools modify applications to insert control information, while non-intrusive ones do not. However, the common goal is to build the causal path of a request.

Tools for building causal paths and, consequently, detecting anomalies were designed for the current Internet architecture. This architecture has been heavily criticized for being difficult to change and evolve, even being labeled as "ossified" by some researchers. The new paradigm of Software Defined Networking (SDN) was proposed to overcome the problems of the current Internet architecture. SDN provides a decoupling between control and data planes, allowing external software elements to perform functions of the control plane and to modify the behavior of the data plane.

This paper proposes S-Trace, a non-intrusive tool for building causal paths that

explores the positive features of intrusive and non-intrusive tools for anomaly detection and the separation of planes provided by SDN. S-Trace intercepts library function calls to correlate Inter-Process Communication (IPC) events used to process a request. Moreover, S-Trace explores the control and data plane decoupling of SDN to replay network traffic for improving the causal paths constructed.

The evaluation results show that S-Trace builds correct causal paths which capture the behavior of the applications during the processing of a request. The evaluation uses two applications that cover a large spectrum of implementation models used for building distributed applications. During evaluation, S-Trace was able to build the causal paths even with fault injection and in the presence of techniques that modify the IP addresses of a connection like NAT, issues that make the building process much harder.

Keywords: *causal path, software defined networking, SDN, distributed applications.*

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Siglas

API *Application Programming Interface*

DNS *Domain Name System*

EJB *Enterprise JavaBeans*

GDS *Grafo de Dependências de Serviços*

GI *Grafo de Inferência*

HP *Hewlett-Packard*

HTTP *Hypertext Transfer Protocol*

IAC *Interface entre as camadas de Aplicação e de Controle*

ICI *Interface entre as camadas de Controle e de Infraestrutura*

IP *Internet Protocol*

IPC *Inter-Process Communication*

J2EE *Java Plataform, Enterprise Edition*

JDBC *Java Database Connectivity*

LWPID *Lightweight Process ID*

PID *Process ID*

SDN *Software Defined Network*

SQL *Structured Query Language*

SSL *Secure Socket Layer*

TCP *Transmission Control Protocol*

URL *Uniform Resource Locator*

Agradecimentos

Aos meus pais e família por todo apoio recebido. Agradeço por proporcionar todo o âmbito necessário para realizar um trabalho de qualidade, além dos diversos incentivos para continuar em direção aos meus objetivos apesar das inúmeras dificuldades.

Ao professor Ronaldo Alves Ferreira pela orientação tanto acadêmica quanto pessoal. Agradeço pela paciência, pela confiança e pelos excelentes conselhos transmitidos que levarei por toda vida.

Aos professores Hana Karina Rubinsztejn e Luciano Gonda pelas correções e comentários sugeridos na qualificação e na defesa. Ao professor Fábio Moreira Costa por aceitar o convite de participação da banca de defesa e por colaborar com as devidas correções.

À Faculdade de Computação (FACOM) da Universidade Federal de Mato Grosso do Sul por fornecer um excelente ambiente de formação acadêmica tanto de infraestrutura quanto de ensino. Aos professores da FACOM por transmitir, diretamente ou indiretamente, o conhecimento base para o desenvolvimento da minha formação acadêmica.

À CAPES pelo suporte financeiro.

Capítulo 1

Introdução

Centros de processamento de dados modernos hospedam milhares de servidores para prover serviços de Internet de larga escala, como: DNS e LDAP, por exemplo. Normalmente, cada servidor executa um conjunto diverso de aplicações de rede. Grandes empresas, como a HP (*Hewlett-Packard*), possuem mais de 6000 diferentes aplicações em uso em um único centro de dados [14]. As interações nesse ambiente — sejam entre aplicações ou entre dispositivos ou entre aplicações e dispositivos — se tornam extremamente complexas a medida que novos dispositivos ou aplicações são adicionadas aos centros de dados, tais como: dispositivo de balanceamento de carga, uma aplicação *Web*, etc.

A partir das interações, as dependências entre as aplicações e os dispositivos se tornam evidentes. Uma aplicação pode ser dependente de outra e de dispositivos de hardware. Por exemplo, uma simples requisição de um navegador a um endereço *Web* possui, pelo menos, duas dependências: a aplicação do serviço de *Domain Name System* (DNS), que resolve o nome, e a aplicação do serviço *Web* que retorna a página ao navegador. Além disso, essa requisição possui diversas dependências de dispositivos de rede, como: enlaces, comutadores, balanceadores de carga, entre outros.

1.1 Caminho Causal

O caminho causal de uma requisição é o conjunto de todos os componentes, tanto de hardware como de software, utilizados para processá-la. Um caminho causal bem definido auxilia a identificação dos componentes que estão apresentando problemas, caso a requisição apresente um comportamento não esperado. Falhas, sobrecargas ou erros em elementos do sistema de comunicação são denominados anomalias. Enlaces congestionados, roteadores mal configurados, ou até mesmo falhas nas máquinas que processam a requisição são exemplos típicos de anomalias de rede.

Atualmente, existem diversas ferramentas de monitoramento conceituadas, como Nagios [2], Icinga [3] e Zenoss [4], que auxiliam a detectar anomalias de rede. Entretanto, essas ferramentas necessitam de operadores com um extenso conhecimento da rede e das aplicações que são monitoradas. Normalmente, elas fornecem apenas informações de conectividade ou de disponibilidade.

O processo de construir caminhos causais e de detectar anomalias tem sido amplamente estudado pela academia e pela indústria, fornecendo diversas ferramentas, como: Sherlock [34], X-Trace [35], BorderPatrol [37], entre outras. Essas ferramentas são divididas em dois grandes grupos: as intrusivas e as não intrusivas. A principal diferença entre os grupos é que nas intrusivas as aplicações precisam ser modificadas para se inserir informações de controle, enquanto nas não intrusivas não há essa necessidade.

1.2 Rede Definida por Software

A maioria das ferramentas de construção de caminhos causais e de detecção de anomalias até então conhecidas foram concebidas para a arquitetura atual da Internet. Essa arquitetura tem sido fortemente criticada por ser de difícil alteração e evolução, sendo inclusive rotulada de ossificada. O novo paradigma de *Rede Definida por Software* (SDN – *Software Defined Network*) foi proposto para contornar os problemas da arquitetura atual da Internet. SDN proporciona a dissociação entre o plano de controle e o plano de dados, permitindo que elementos externos de software exerçam funções do

plano de controle e alterem o comportamento do plano de dados. A partir dessa dissociação, a arquitetura das SDNs permite que a inteligência da rede seja concentrada em um único ponto. Esse ponto possui uma visão global da rede e torna a gerência uma tarefa mais simples.

1.3 Objetivos e Contribuições

O objetivo deste trabalho é a construção e o projeto de S-Trace, uma ferramenta não intrusiva de construção de caminhos causais que explora aspectos positivos das ferramentas de detecção de anomalias intrusivas e não intrusivas, e a separação de planos fornecida pelo paradigma das SDNs. As ferramentas de detecção proporcionam diferentes técnicas próprias de construção de caminhos causais que S-Trace explora e abstrai nos caminhos causais construídos. Esses caminhos auxiliam as detecções das ferramentas de detecção de anomalias e os administradores de rede a entender o comportamento das aplicações.

Atualmente, existem diversas aplicações de rede que são implementadas utilizando diferentes técnicas de implementação visando otimizar, principalmente, o desempenho no processamento das requisições. Com isso, as interações em centros de dados que possuem essas aplicações se tornam muito complexas. Ferramentas de construção de caminhos causais que não necessitam conhecer a arquitetura interna das aplicações e nem modificá-las são potenciais ferramentas a serem utilizadas nesses centros de dados. Nesse cenário, S-Trace se torna uma interessante ferramenta, visto que não modifica as aplicações e não faz nenhuma hipótese sobre o funcionamento dessas aplicações para construir os caminhos causais auxiliando os administradores de rede a compreender as interações das aplicações.

Para construir os caminhos causais, S-Trace observa os mecanismos de comunicação entre processo (IPC – *Inter-Process Communication*) utilizados nas trocas de mensagens de aplicação. Essa observação é realizada por meio de interceptação de chamadas de função invocadas por *threads* e processos. S-Trace combina os eventos de IPC, através de

identificadores comuns, para construir os caminhos causais das requisições. Além disso, a fim de reduzir a sobrecarga nas estações de trabalho e servidores, S-Trace utiliza o conceito de agente e gerente, em que os agentes são instalados nas máquinas a serem monitoradas para enviar as informações dos eventos de IPC e os cabeçalhos dos pacotes de rede para o gerente da ferramenta. A partir dessas informações, o gerente inicia o processo de construção de caminhos causais. S-Trace explora a separação de planos fornecida pelas SDNs, mais especificamente, utiliza as mensagens dos planos de controle e do plano de dados armazenadas para realizar a reprodução de tráfego de rede, a fim de aprimorar os caminhos causais construídos. Essa reprodução reinjeta os cabeçalhos dos pacotes de rede, a fim de observar o comportamento da rede ao receber os pacotes.

S-Trace foi avaliada utilizando duas aplicações representativas dos principais modelos de implementação. A primeira aplicação emula o comportamento de uma aplicação *Web* multicamadas (*multi-tier*) que realiza comunicações entre múltiplas *threads* e processos em diversos níveis de profundidade. Ela utiliza diferentes mecanismos de comunicação entre processos e foi instrumentada para mostrar o caminho causal exato de uma requisição. O objetivo principal dessa aplicação é comparar os resultados gerados por S-Trace com os caminhos causais exatos gerados pela instrumentação da aplicação. A segunda aplicação é um *benchmark* bem conhecido, TPC-W [20], utilizado para emular aplicações de comércio eletrônico (*e-commerce*). Os resultados da avaliação mostram que S-Trace é capaz de construir caminhos causais precisos, mesmo na presença de falhas e de dispositivos NAT que modificam os endereços IP das conexões utilizadas nas requisições. Falhas e mecanismos de tradução de endereço dificultam significativamente a construção de caminhos causais.

1.4 Organização do Texto

Este trabalho está organizado como segue. O Capítulo 2 apresenta os conceitos básicos do domínio do problema. Uma breve introdução ao paradigma SDN e uma ferramenta de reprodução de tráfego utilizada neste trabalho são apresentadas no Capítulo 3. O Capítulo 4 discute as principais ferramentas utilizadas para construção de caminhos

causais existentes na literatura. Os principais módulos de S-Trace, bem como os algoritmos desenvolvidos neste trabalho, são descritos no Capítulo 5. O Capítulo 6 apresenta as avaliações e os resultados experimentais. Conclusões e trabalhos futuros são apresentados no Capítulo 7.

Capítulo 2

Conceitos Básicos

Centros de processamento de dados utilizam uma ampla variedade de serviços e aplicações de rede que são, normalmente, dependentes entre si. A utilização de equipamentos de rede como balanceadores de carga, réplicas e *caches* torna a captura de dependências entre as aplicações e os equipamentos de rede uma tarefa extremamente desafiadora, devido às diversas interações entre aplicações e serviços.

Uma requisição de uma aplicação pode trafegar em diversos elementos de rede para ser processada. Determinar corretamente esses elementos, tanto de software quanto de hardware, é uma tarefa essencial para as ferramentas de detecção de anomalias. O conjunto de elementos de rede, software ou hardware, utilizados para processar a requisição é definido como caminho causal. Dessa forma, uma falha em um elemento do caminho afeta o processamento da requisição.

Em uma aplicação concorrente, uma requisição pode ser processada em múltiplas *threads* ou processos, fazendo com que eventos relacionados a uma mesma requisição sejam de difícil associação. Além disso, há diversas formas de se implementar uma aplicação concorrente, como múltiplos processos, múltiplas *threads* ou eventos assíncronos. Dessa forma, obter caminhos causais de aplicações concorrentes não é uma tarefa trivial.

Por se tratarem de conceitos fundamentais para o entendimento deste trabalho, caminhos causais e formas de implementação de aplicações concorrentes são discutidos

nas seções subsequentes.

2.1 Caminho Causal

Um caminho causal é definido como o conjunto de elementos de rede — hardware e software — que são utilizados no processamento de uma requisição, por exemplo, enlaces, comutadores e aplicações. Ele é o principal artefato utilizado pelas ferramentas de detecção de anomalias, visto que os elementos desse conjunto podem sofrer falhas ou sobrecargas, afetando o comportamento da aplicação.

O caminho causal pode possuir diversos níveis de granularidade, dependendo da ferramenta que o utiliza. Sherlock [34], por exemplo, inclui em seus caminhos causais desde enlaces de comunicação e dispositivos de comutação até as aplicações que processam as requisições. Por outro lado, BorderPatrol [37] utiliza apenas informações das aplicações para a construção dos caminhos causais. Sherlock e BorderPatrol são ferramentas de detecção de anomalias discutidas nas Seções 4.2.1 e 4.2.2, respectivamente.

As Figuras 2.1 e 2.2 apresentam exemplos de caminhos causais de uma mesma requisição HTTP em dois níveis de granularidade no seguinte cenário: um usuário requisita uma página *Web* em seu navegador; a requisição é recebida e processada por um servidor *proxy* que a repassa para o destino final, o servidor *Web* que contém a página. A Figura 2.1 ilustra um caminho causal de granularidade grossa e a Figura 2.2 ilustra o mesmo caminho causal, porém com granularidade fina.

Figura 2.1: Exemplo de caminho causal de granularidade grossa.

O caminho de granularidade grossa mostra que a requisição foi processada por três

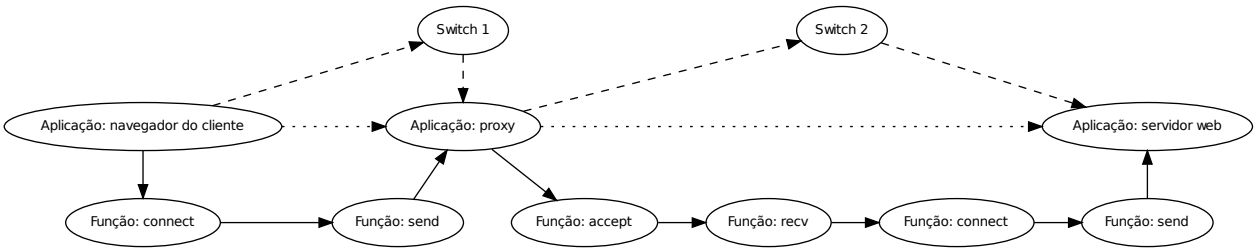


Figura 2.2: Exemplo de caminho causal de granularidade fina.

aplicações, o navegador, que originou a requisição, o proxy e o servidor *Web*. Por outro lado, o caminho de granularidade fina mostra que a requisição foi processada pelas mesmas três aplicações, porém, mostra quais foram as chamadas de função invocadas por cada aplicação e os comutadores utilizados para trafegar a requisição. Com isso, é possível determinar em que ponto o processamento apresentou um comportamento não esperado, por exemplo, qual função levou mais tempo para processar, em qual função houve algum erro, por qual comutador a requisição foi trafegada, etc.

A construção de caminhos causais é uma das tarefas mais importantes para as ferramentas de detecção de anomalias. Caminhos causais bem definidos facilitam a detecção, pois contêm todos os componentes envolvidos no processamento das requisições. Dessa forma, se uma requisição apresentou um comportamento não esperado, então o componente responsável está contido no caminho causal dessa requisição. Entretanto, a construção de caminhos causais não é uma tarefa trivial, devido às técnicas de implementação utilizadas pelas aplicações, à presença de diferentes equipamentos de rede, aos mecanismos de comunicação entre os processos e *threads*, e à arquitetura em que as aplicações estão inseridas, entre outros.

2.2 Modelos de Implementação de Aplicações de Rede

Cada aplicação de rede pode ser implementada de diversas maneiras e a escolha depende de fatores como: desempenho, tamanho da aplicação, quantidade de requisições simultâneas que devem ser processadas, familiaridade do programador, etc. Portanto, a escolha do

modelo a ser utilizado para a implementação deve ser realizada de maneira cuidadosa.

O modelo de implementação utilizado pela aplicação afeta as ferramentas de construção de caminhos causais, pois uma única requisição pode ser processada por diversos elementos do sistema operacional, mais especificamente, diversas *threads* de um único processo ou de vários processos. Além disso, as *threads* que processam a requisição podem realizar novas requisições para outras aplicações ou serviços. Consequentemente, uma falha em uma das *threads*, ou nas novas requisições, afeta diretamente o processamento de toda a requisição.

As ferramentas de construção de caminhos causais devem capturar todas as interações que são realizadas pelas *threads* ao receber uma requisição, independentemente do modelo escolhido para a implementação das aplicações. Os principais modelos de implementação de aplicações de rede utilizam múltiplos processos, *threads* ou eventos assíncronos [26]. As próximas seções apresentam uma breve revisão sobre esses modelos.

2.2.1 Processos

Um processo é uma instância de um programa que está sendo executado pelo sistema operacional [28]. Cada processo possui um espaço exclusivo na memória que é dividido em pilha, *heap*, código objeto e dados. Os processos são criados e destruídos pelo sistema operacional, mais especificamente por processos já existentes, utilizando as funções `fork` e `kill`, respectivamente. Porém, a criação demanda tempo para ser realizada devido aos recursos que são alocados para o novo processo.

Os processos são independentes ou cooperativos. Os processos independentes não são afetados e não afetam outros processos do sistema. Por outro lado, os cooperativos precisam trocar dados entre si, essas trocas são realizadas por meio de mecanismos de comunicação inter-processo (IPC – *Inter-Process Communication*). As ferramentas de construção de caminhos causais observam, principalmente, os processos cooperativos, visto que uma única requisição pode ser processada por vários processos. Para isso, essas ferramentas devem capturar a forma como as informações são trocadas entre os processos. Sistemas operacionais modernos fornecem diversos mecanismos de IPC, como: *socket*, fila

de mensagem, *pipe* e memória compartilhada.

A forma mais usual de se implementar uma aplicação concorrente com processos é criar um novo processo para cada nova conexão, dessa forma, um único processo fica no estado de bloqueado, para receber as novas conexões. A cada conexão estabelecida, um novo processo é criado para processar as requisições dessa conexão. Contudo, devido ao tempo gasto na criação de um novo processo e aos recursos alocados, essa forma de implementação é muito onerosa e lenta.

2.2.2 *Threads*

De forma semelhante a um processo, uma *thread* é uma unidade básica de um programa em execução. Em sistemas operacionais modernos, *threads* estão inseridas no interior de um processo e compartilham memória e recursos alocados para o processo, como: sinais, descritores de arquivos e *sockets*.

Os principais benefícios da utilização de *threads* são volume de resposta, economia e compartilhamento de recursos. O uso de *threads* em uma aplicação com *multithreading* faz com que a execução da aplicação prossiga normalmente enquanto algumas *threads* da aplicação estão bloqueadas ou lentas, aumentando o volume de resposta da aplicação. *Threads* compartilham recursos e memória de um processo, dessa forma, a criação de *threads* não é uma operação onerosa e lenta, pois não realiza alocação de novos recursos, diferentemente da criação de processos.

Portanto, uma outra forma de se implementar uma aplicação concorrente, de forma equivalente ao explicado na Seção 2.2.1, é utilizando *threads* no lugar de processos. Dessa forma, o processamento é mais rápido e menos oneroso, pois não há criação de um novo processo e, consequentemente, alocação de recursos para cada nova conexão.

2.2.3 *Pool* de Processos/*Threads*

Nas técnicas mais triviais de implementação de aplicações, processos ou *threads* são criados após o estabelecimento de novas conexões. Dessa forma, o tempo de criação é agregado

ao tempo de processamento da requisição. A fim de reduzir esse tempo agregado, diversas técnicas que exploram diferentes formas de uso de processos e *threads* foram propostas, dentre as quais as técnicas de *pool* de processos ou *pool* de *threads* [26, 27] são as mais utilizadas.

Na técnica de *pool*, uma determinada quantidade de processos ou *threads* é criada e organizada em uma fila no início da execução da aplicação. A cada conexão estabelecida, um elemento dessa fila é escolhido para processar as requisições da conexão. Dessa forma, o tempo de criação dos processos ou das *threads* não é agregado ao tempo de processamento, pois, a criação já foi realizada no início da execução da aplicação.

2.2.4 `Select`

Em ambientes Unix, há cinco diferentes modelos de entrada e saída [26] para a implementação de aplicações. Esses modelos são: bloqueantes, não bloqueantes, multiplexados, dirigidos por sinais e assíncronos. Normalmente, os três primeiros modelos são os mais utilizados em aplicações de rede.

No modelo bloqueante, o processo ou a *thread* é bloqueado até que um determinado evento ocorra. Por outro lado, no modelo não bloqueante, caso o evento não ocorra, a aplicação deve continuamente repetir a operação. Essa repetição é denominada *polling*.

A função `select` faz parte dos modelos multiplexados. Ela solicita ao *kernel* que notifique o processo que a invocou quando determinados eventos ocorrerem ou ao fim de um determinado tempo. A notificação é realizada quando descritores de arquivos específicos estiverem prontos para operações de entrada e saída. Dessa forma, processos ou *threads* podem realizar outras operações enquanto estiverem esperando por eventos. Essa capacidade é dificilmente obtida por meio de modelos bloqueantes ou não bloqueantes.

Seguir o fluxo de dados de uma aplicação que utiliza a função `select` para processar uma requisição e, consequentemente, construir caminhos causais, é uma tarefa extremamente desafiadora, pois o fluxo depende de como programadores a utilizam e quais mecanismos de IPC foram envolvidos nas trocas de mensagens entre processos e

threads.

2.3 Considerações Finais

As aplicações de rede estão em constante evolução no que tange a implementação. Diversos modelos de implementação podem ser utilizados para que o processamento das requisições seja mais eficaz e eficiente. Esse processamento envolve diversos elementos do sistema operacional e da rede, como: processos, *threads*, enlaces e comutadores. Todos esses elementos estão contidos em um conjunto denominado caminho causal da requisição.

Este capítulo define caminhos causais e seus elementos, além dos possíveis tipos de granularidade, e revisa, brevemente, os principais modelos de implementação de aplicações de rede. S-Trace utiliza esses conceitos para construir os caminhos causais das requisições.

Capítulo 3

Redes Definidas por Software

3.1 Introdução

Redes Definidas por Software (SDNs) constituem um novo paradigma para pesquisa e desenvolvimento na área de redes de computadores e vêm se tornando foco de pesquisa em grande parte da indústria e da academia. Esse paradigma surgiu com o objetivo de retirar o rótulo de "ossificada" da arquitetura atual da Internet, que é considerada de difícil gerenciamento e pouco flexível para introdução de novos protocolos e tecnologias.

SDNs preconizam a dissociação entre o plano de controle e o plano de dados em uma rede de computadores e centralizam a inteligência da rede em um único ponto lógico. A centralização é realizada por meio de controladores SDN baseados em software que manipulam mecanismos de encaminhamento dos dispositivos de comutação da rede por meio de uma interface de programação bem definida fornecida pelos dispositivos. Dessa forma, administradores de rede, por meio dos controladores, possuem um controle central programável da rede e, com isso, não precisam acessar os dispositivos individualmente para configurá-los.

Devido à importância dos controladores no novo paradigma das SNDs, diversas implementações podem ser encontradas na literatura, como: NOX [22], Beacon [45], FlowVisor [24] e Maestro [25], entre outros.

A centralização lógica do estado da rede pelos controladores permite que os administradores da rede possam gerenciar, configurar, manipular e otimizar recursos da rede de modo autônomo sem solicitar aos fabricantes dos equipamentos de rede, por meio dos controladores. Isto difere das redes atuais em que novas funcionalidades nos dispositivos de rede devem ser solicitadas aos fabricantes. A disponibilização de novas funcionalidades é um processo moroso e que gera custos para as empresas com a atualização dos equipamentos.

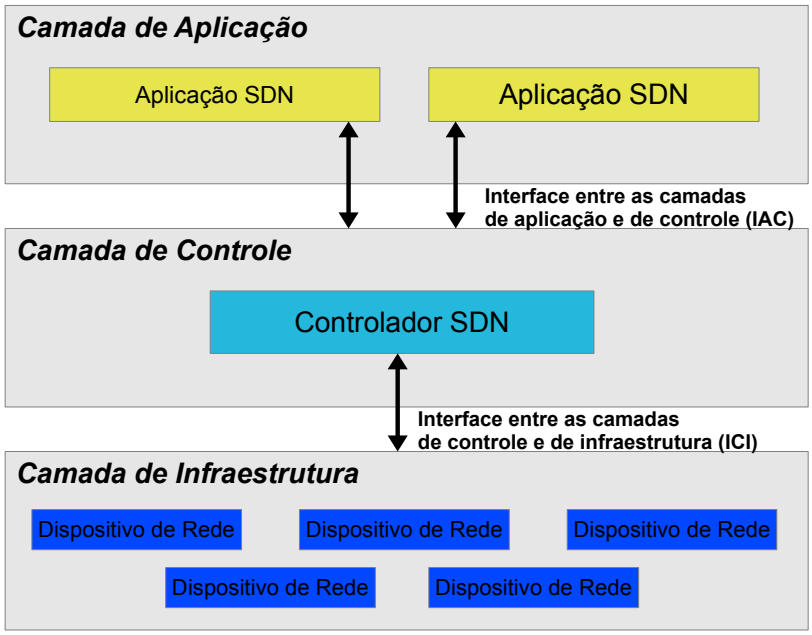


Figura 3.1: Exemplo de uma arquitetura de rede que segue o paradigma das SDNs [19].

SDNs necessitam de equipamentos mais simples visto que eles devem apenas receber e atender as instruções fornecidas pelos controladores por meio da interface de controle. A Figura 3.1 ilustra uma arquitetura básica de rede que segue o paradigma das SDNs e seus componentes. A camada de aplicação abrange a implementação utilizada pelas aplicações dos controladores, facilitando a administração e gerenciamento da rede pelos administradores. A camada de controle compreende a implementação de baixo nível dos controladores, que recebe mensagens das aplicações em alto nível e as traduzem para instruções de baixo nível de tal forma que os dispositivos de rede as reconheçam e as processem. Já a camada de infraestrutura engloba os dispositivos físicos de rede.

A arquitetura básica de SDN possui duas interfaces de comunicação, Interface entre as camadas de Aplicação e de Controle (IAC) e Interface entre as camadas de Controle e de Infraestrutura (ICI). A IAC é a interface entre as camadas de aplicação e de controle, onde as mensagens em alto nível das aplicações são repassadas para os controladores. ICI é a interface entre as camadas de controle e de infraestrutura, onde as instruções são traduzidas para mensagens de baixo nível para os dispositivos de rede. O protocolo mais utilizado na interface ICI é o OpenFlow [21].

3.2 OpenFlow

OpenFlow foi o primeiro protocolo de comunicação entre comutadores e controladores. Ele permite acesso direto aos modos de encaminhamento dos comutadores por meio de uma interface padronizada. Dessa forma, a rede é controlada por meio de controladores baseados em software.

Porta de entrada comutad.	Origem Ethernet	Destino Ethernet	Tipo Ethernet	Ident. VLAN	Priorid. VLAN	Rótulo MPLS	Classe de tráfego MPLS	Origem IPv4	Destino IPv4	Protoc. IPv4 ou Código ARP	Bits ToS IPv4	Porta de origem TCP/UDP/SCTP ou Tipo ICMP	Porta de destino TCP/UDP/SCTP ou Código ICMP

Figura 3.2: Campos de cabeçalho dos pacotes utilizados para a definição de fluxo OpenFlow (Versão *1.1*.0).

OpenFlow utiliza o conceito de fluxos para a manipulação do plano de dados de um dispositivo. Um fluxo é constituído pela junção de campos dos cabeçalhos dos pacotes de rede. Dessa forma, diversos níveis de fluxos podem ser definidos, desde fluxos específicos até fluxos mais genéricos, utilizando mais ou menos campos, respectivamente. A Figura 3.2 ilustra os campos utilizados para a definição de um fluxo de acordo com a Versão *1.1*.0 do protocolo OpenFlow [21]. Os campos descritos na figura são os mesmos dos protocolos Ethernet, IP, TCP, UDP e ICMP.

Os equipamentos OpenFlow possuem três componentes: tabela de fluxos, canal seguro e protocolo OpenFlow. A tabela de fluxos contém entradas que determinam como processar os fluxos. Cada entrada é denominada como regra de fluxo e consiste na definição do fluxo, contadores e ações a serem tomadas. Os contadores armazenam

informações sobre os fluxos, como quantos pacotes trafegaram naquele fluxo, quantidade de bytes, etc. As possíveis ações que o dispositivo pode tomar são: encaminhar para uma porta específica, alterar campos dos cabeçalhos dos pacotes, descartar e/ou encaminhar para o controlador. A Figura 3.3 mostra o formato da tabela de fluxos e suas entradas para um dispositivo OpenFlow.

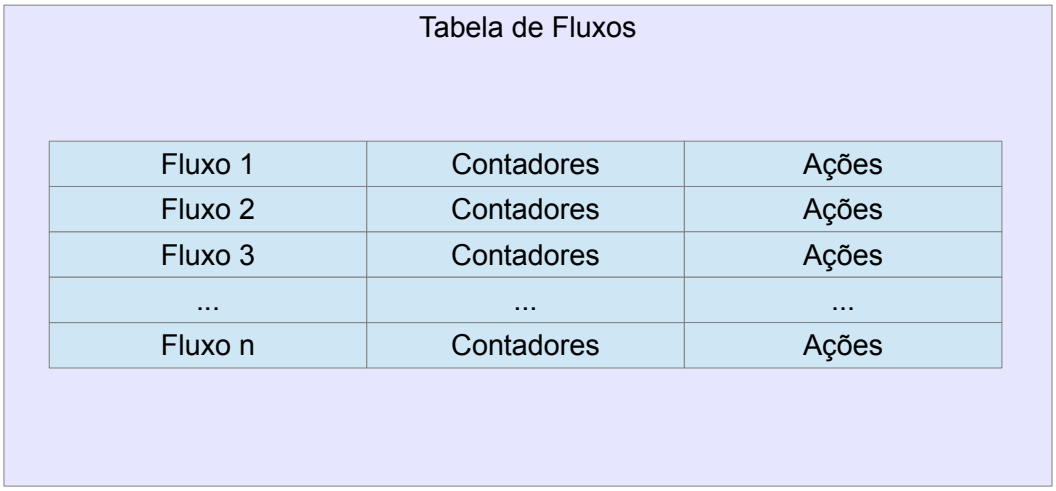


Figura 3.3: Tabela de fluxos de um dispositivo OpenFlow e suas entradas.

Um dispositivo OpenFlow é manipulado por um controlador que realiza as operações por meio de um canal seguro de comunicação utilizando o protocolo OpenFlow. Essas operações são: adição, modificação e remoção de regras de fluxo; obtenção dos valores de contadores de regras; extração de informações dos dispositivos. O canal seguro faz com que a comunicação não sofra ataques mal intencionados. Para isto, o protocolo criptográfico *Secure Socket Layer* (SSL) tem sido o mais utilizado. A Figura 3.4 mostra a arquitetura geral de um *switch* OpenFlow simples e as ligações com o Controlador e os *hosts*. Nessa figura, é possível observar que o *switch* está ligado a quatro *hosts* e ao controlador e possui uma tabela de fluxos. A ligação entre o *switch* e o controlador é feita por meio do canal seguro com SSL, sobre o qual é executado o protocolo OpenFlow.

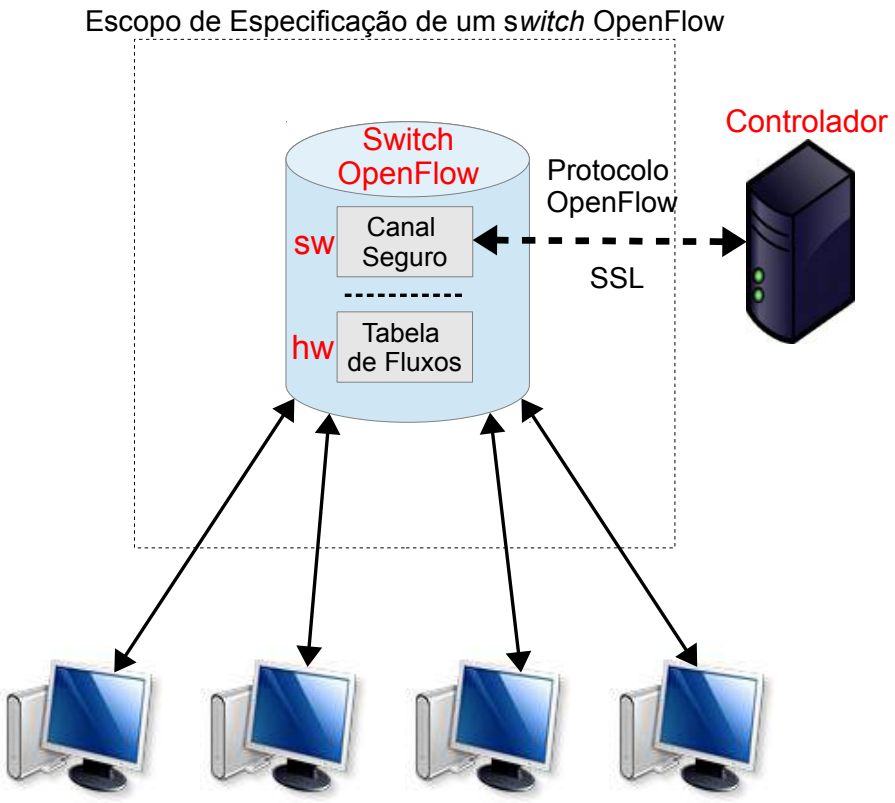


Figura 3.4: Arquitetura geral de um *switch* OpenFlow simples [21].

3.3 OFRewind

OFRewind [43] é uma ferramenta para depuração de redes programáveis que consolida as técnicas de gravação e de reprodução de tráfegos de rede em um ambiente programável. Essa ferramenta baseia-se em arquiteturas programáveis de separação de planos, como OpenFlow [21], para realizar a gravação e a reprodução dos tráfegos de rede. A partir dessas técnicas, OFRewind é capaz de reproduzir erros de software, identificar limitações nos caminhos dos dados e localizar erros de configuração. Para implementar essas técnicas, a ferramenta é dividida em dois componentes principais, OFRewind e DataStore.

O componente OFRewind da ferramenta armazena e reproduz as mensagens do plano de controle trocadas entre os comutadores e o controlador. Essas mensagens são importantes para orquestrar a reprodução dos tráfegos do plano de dados, uma vez que a reprodução dos pacotes depende diretamente das regras instaladas nos comutadores. Se um pacote for reinjetado na rede em uma situação em que as regras nos comutadores

não sejam exatamente iguais às regras de fluxo instaladas durante a gravação, o comportamento da rede não é reproduzido fielmente, fazendo com que essa ferramenta de depuração não seja eficaz. O armazenamento das mensagens do plano de controle é realizado interceptando essas mensagens oriundas do controlador antes de repassá-las aos comutadores. Portanto, o componente OFRewind atua como um *proxy*, interceptando mensagens trocadas entre o controlador e os dispositivos de rede.

DataStore armazena e reinjeta as mensagens do plano de dados que trafegam nos comutadores, mais especificamente, os pacotes de rede. Esse componente é conectado diretamente aos comutadores presentes na rede em que o tráfego deve ser observado e armazenado. Quando um determinado fluxo deve ser armazenado, OFRewind instrui certos comutadores para que os fluxos também sejam repassados ao DataStore, além de seu destino original. Na reprodução, OFRewind realiza a comunicação com o DataStore fazendo com que este reinjete os pacotes de maneira coordenada. A Figura 3.5 ilustra um exemplo de um ambiente com a ferramenta OFRewind. Na figura, é possível observar que os *switches* estão ligados diretamente ao componente OFRewind e este, por sua vez, ligado ao controlador. Dessa forma, qualquer mensagem do plano de controle oriunda do controlador deve ser repassada primeiramente para o componente OFRewind. Além disso, apenas dois *switches* estão ligados a um DataStore cada. Com isso, as mensagens dos planos de dados desses *switches* são armazenadas nos DataStores ligados a cada *switch*.

Cada componente da ferramenta é dividido em dois subcomponentes: OFRewind em OFRecord e OFReplay; e DataStore em Datarecord e Datareplay. A Figura 3.6 ilustra a divisão dos componentes da ferramenta OFRewind. Essa divisão visa separar os módulos de gravação e de reprodução de cada plano. O módulo OFRecord captura e intercepta as mensagens de controle entre os comutadores e o controlador e o módulo OFReplay reinjeta essas mensagens, coordenando o ambiente para a reprodução do tráfego do plano de dados. O módulo DataRecord recebe e armazena os pacotes que trafegam pelo comutador ligado ao DataStore e o módulo DataReplay reinjeta os pacotes armazenados.

Para a correta reprodução do tráfego, é necessário um sincronismo entre as mensagens do plano de controle e do plano de dados. Para isso, OFRewind deve ser capaz de reproduzir todos os fluxos na mesma ordem que foram gravados. Caso

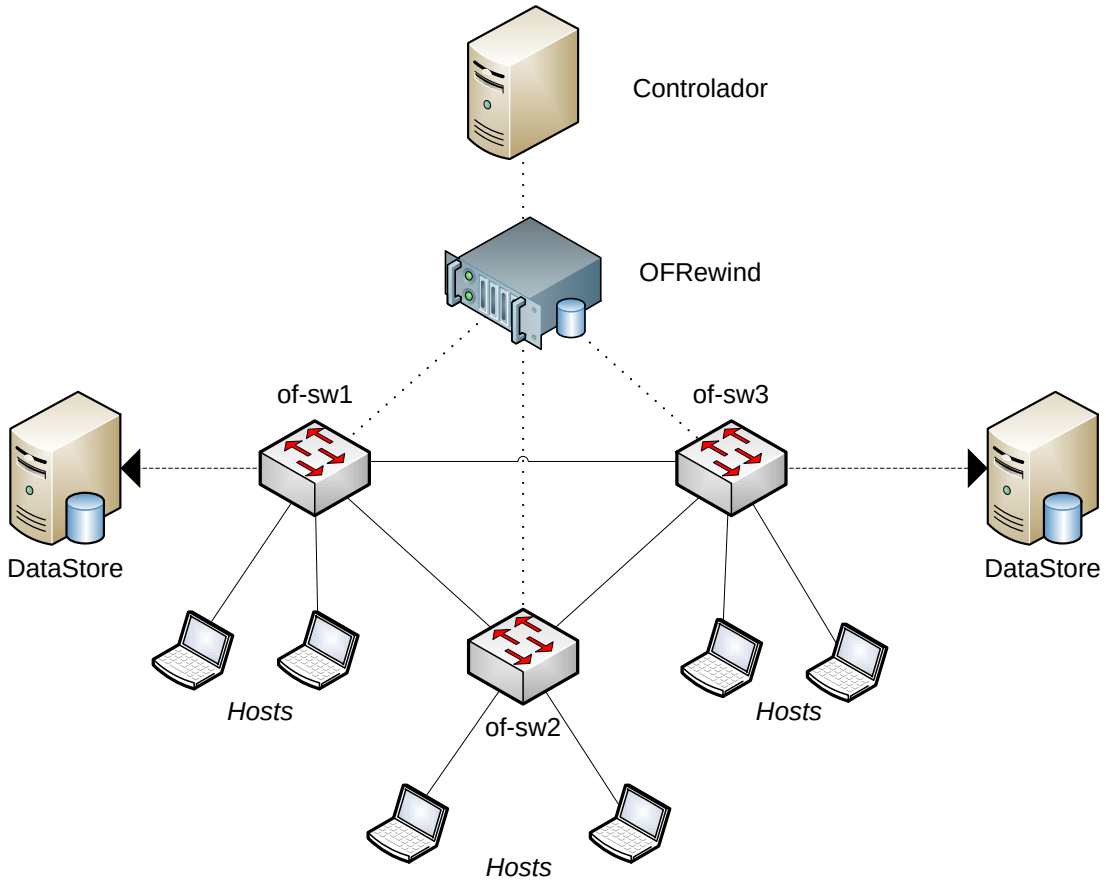


Figura 3.5: Exemplo de um ambiente com a ferramenta OFRewind, em que os DataStores estão ligados somente nos *switches* of-sw1 e of-sw3. Contudo, cada *switch* é ligado ao OFRewind, que por sua vez, é ligado ao Controlador.

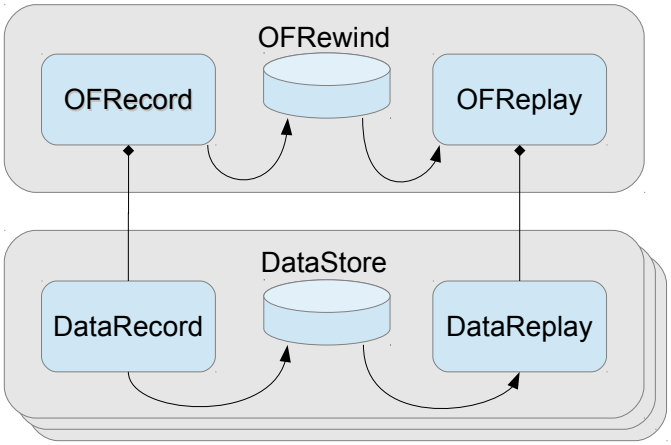


Figura 3.6: Divisão dos componentes da ferramenta OFRewind. Os subcomponentes OFRecord e DataRecord produzem uma coleção de dados que são usados pelos subcomponentes OFReplay e DataReplay, respectivamente. Os subcomponentes do OFRewind armazenam mensagens do plano de controle e os subcomponentes do DataStore armazenam do plano de dados.

contrário, o resultado da reprodução pode não ser o esperado. Por exemplo, se uma regra não foi instalada na tabela de fluxos de um *switch*, este pode rejeitar um determinado fluxo por não estar definido em sua tabela de fluxo, devido à falta da reprodução da mensagem do plano de controle. A Figura 3.7 ilustra exemplos de uma sincronização mal sucedida e outra com sucesso.

A Figura 3.7 (a) ilustra um exemplo em que a reprodução não foi sincronizada de maneira correta. Dessa forma, o pacote reinjetado pelo DataStore, ao chegar ao *switch*, é descartado, visto que a regra de fluxo armazenada pelo OFRewind não foi instalada no momento correto. Com isso, a reprodução não é realizada de maneira correta, não reproduzindo fielmente o comportamento da rede durante a gravação.

A Figura 3.7 (b) ilustra um exemplo em que a reprodução foi sincronizada com sucesso e o pacote reinjetado chega ao seu destino final, o Servidor. O encaminhamento é realizado corretamente, pois a regra de fluxo foi instalada no momento correto pelo OFRewind, fazendo com que o pacote não seja descartado.

Os casos de uso apresentados em [43] demonstram os benefícios da ferramenta OFRewind em diversos casos práticos, nos quais ela permite a localização dos problemas

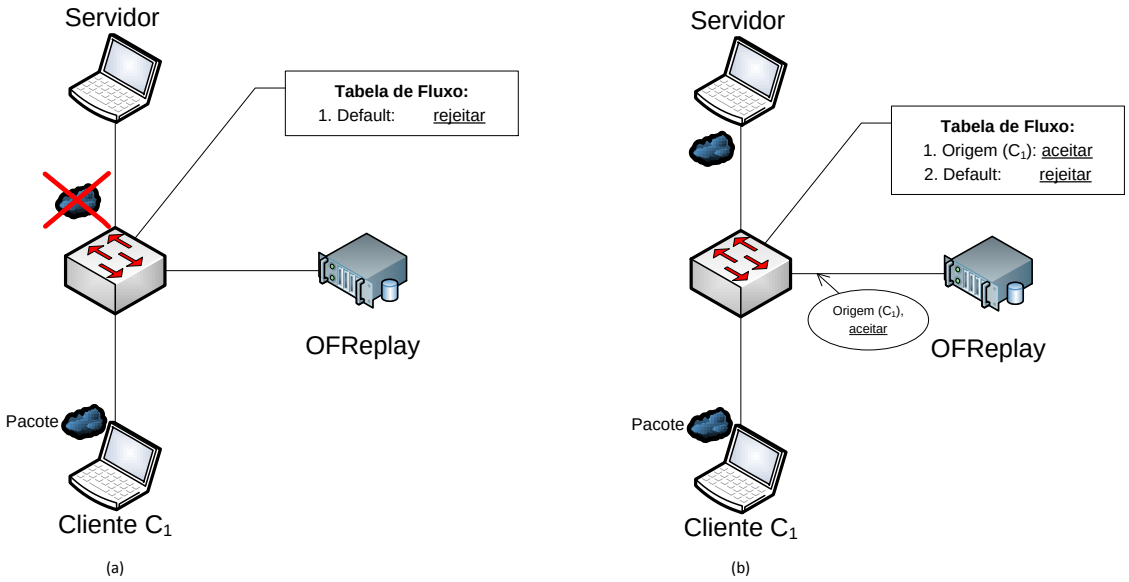


Figura 3.7: (a) Exemplo de uma reprodução mal sincronizada, em que o pacote é descartado devido à falta da regra de fluxo correta. (b) Exemplo de uma reprodução sincronizada corretamente, em que o pacote é encaminhado ao seu destino correto.

em dispositivos *black box*, controladores e componentes de softwares. Além disso, foram avaliados diversos aspectos da ferramenta, como: escalabilidade, desempenho e sincronização. Com isso, a ferramenta de depuração de rede OFRewind identifica as limitações nos caminhos das mensagens do plano de dados e localiza erros dos sistemas de software que manipulam as mensagens do plano de controle.

3.4 Considerações Finais

Este capítulo apresenta o novo paradigma de Redes Definidas por Software (SDNs), que surgiu para retirar o rótulo de "ossificada" da arquitetura atual da Internet. Baseado nesse paradigma, diversos padrões e protocolo foram desenvolvidos, como o OpenFlow [21], que permite acesso direto aos modos de encaminhamentos dos dispositivos. Como o comportamento da rede é definido por controladores baseados em software, ferramentas de depuração são ferramentas inevitáveis em centros de dados modernos com grande quantidade de dispositivos de rede. S-Trace utiliza as técnicas de gravação e reprodução

do tráfego de rede de OFRewind para realizar as construções de caminhos causais bem definidos.

Capítulo 4

Ferramentas de Construção de Caminhos Causais

Caminhos causais são artefatos utilizados para compreender o comportamento de aplicações de rede, pois possuem os elementos de software e hardware utilizados no processamento de uma requisição. Ferramentas de detecção de anomalias utilizam diferentes técnicas para construir os caminhos causais de requisições. Elas são divididas em dois grandes grupos: intrusivas e não intrusivas. Ferramentas intrusivas instrumentam as aplicações e/ou *frameworks* a fim de inserir informações de controle para que possam construir os caminhos causais e, consequentemente, detectar anomalias. Contudo, as modificações necessitam que os códigos-fonte estejam disponíveis e sejam passíveis de modificação. Com isso, as ferramentas intrusivas ficam fortemente limitadas à arquitetura e implementação das aplicações ou dos ambientes de execução, além de exigir conhecimento prévio do software a ser instrumentado.

As ferramentas não intrusivas não instrumentam aplicações, protocolos e nem *frameworks*. Elas apenas utilizam informações referentes às aplicações e ao ambiente como: tempo de processamento, tempo de resposta e tráfego de rede, entre outras, para construir os caminhos causais. Normalmente, elas utilizam correlações temporais ou associação por identificadores comuns para construir os caminhos. Com isso, essas ferramentas não ficam limitadas à arquitetura e implementação das aplicações e nem

necessitam de conhecimento prévio das aplicações e/ou *frameworks* para instrumentação.

Este capítulo descreve quatro ferramentas de detecção de anomalias — duas intrusivas e duas não intrusivas — e como elas realizam a construção dos caminhos causais das requisições. Devido ao escopo deste trabalho, apenas as informações referentes à construção dos caminhos causais são exploradas, omitindo informações sobre a detecção de anomalias. O entendimento dessas ferramentas é importante porque S-Trace explora pontos positivos dessas ferramentas para incoporá-los ao seu processo de construção de caminhos causais.

4.1 Ferramentas Intrusivas

4.1.1 Pinpoint

Pinpoint [31] é uma ferramenta de detecção de anomalias que instrumenta aplicações para que identificadores únicos sejam transportados nas mensagens de aplicação de uma mesma requisição. Os identificadores inseridos pelas aplicações devem ser visíveis por processos ou *threads* de qualquer outra aplicação. A ferramenta fornece essa visibilidade por meio de modificações de protocolos ou de utilização de protocolos com cabeçalhos extensíveis, como HTTP ou SOAP. A propagação de um identificador único em todo o caminho da requisição faz com que a ferramenta seja capaz de mapear corretamente os caminhos causais das requisições. A partir dos caminhos causais construídos, Pinpoint inicia o processo de detecção de anomalias.

A arquitetura da ferramenta, ilustrada na Figura 4.1, é composta por três componentes: traçador, agregador e repositório. O traçador é instalado nas máquinas para obter observações das aplicações, tais como: carimbo de tempo, componentes envolvidos e nome da máquina. O agregador recebe as observações e constrói os caminhos causais a partir dos identificadores nas mensagens de aplicação das requisições. O repositório armazena os caminhos causais e as observações e realiza os processos de detecção, diagnóstico e visualização de anomalias.

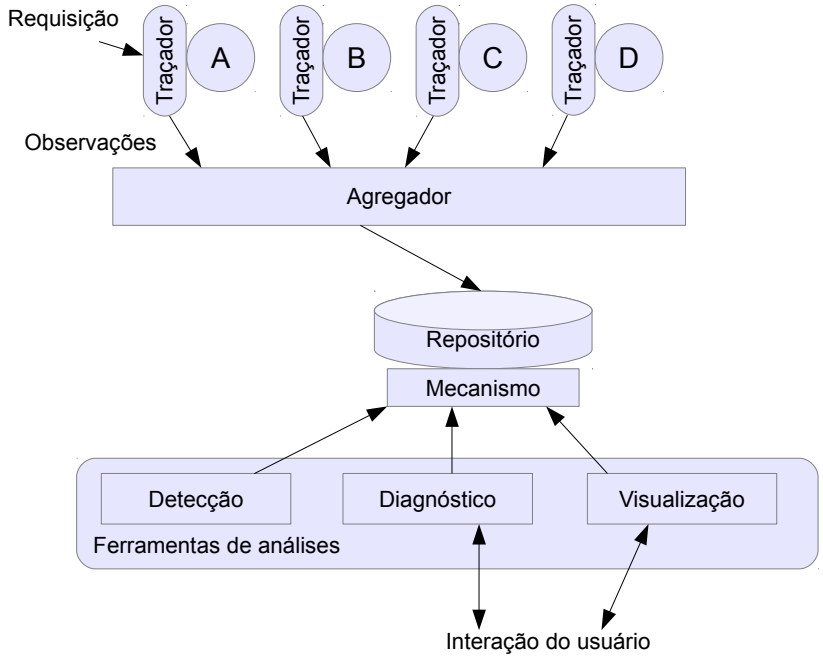


Figura 4.1: Arquitetura de Pinpoint [31].

Os identificadores únicos que são transportados nas mensagens de aplicação garantem um mapeamento correto dos caminhos causais, uma vez que esses identificadores são repassados por todas as aplicações que são utilizadas no processamento da requisição. Dessa forma, mesmo que o processamento de uma requisição necessite realizar diversas outras requisições, todas elas terão o mesmo identificador da requisição inicial. A partir dos identificadores, Pinpoint observa todos os elementos que possuem o mesmo identificador para construir o caminho causal da requisição.

A avaliação de Pinpoint foi realizada por meio da instrumentação do servidor *Web* Jetty [9] para gerar identificadores únicos a cada requisição e inseri-los nas mensagens de aplicação. A ferramenta de *benchmark* TPC-W [20] foi utilizada para simular atividades de um servidor *Web* transacional orientado a comércio. Os caminhos causais construídos por Pinpoint foram plotados em gráficos utilizando GNU Octave [5] e foram traçados utilizando Graphviz [8].

Pinpoint [31] é uma ferramenta de detecção de anomalias que observa as interações

dos componentes ao invés dos componentes individualmente para construir os caminhos causais e detectar as anomalias por meio de testes estatísticos. Embora os resultados de Pinpoint sejam satisfatórios, a instrumentação necessária para a propagação dos identificadores depende do conhecimento prévio das aplicações e da capacidade dos programadores em instrumentar diferentes aplicações para carregarem os identificadores nas mensagens de aplicação, além de demandar muito tempo.

4.1.2 X-Trace

X-Trace [35] é uma das ferramentas de detecção de anomalias mais intrusivas da literatura, pois instrumenta não só as aplicações como toda a pilha de protocolos de rede. A instrumentação tem o objetivo de propagar identificadores nas mensagens trocadas entre as aplicações e as camadas de rede. A técnica de propagação é semelhante à da ferramenta Pinpoint [31].

A ideia básica de X-Trace é propagar metadados (identificadores) nas mensagens de aplicação trocadas para processar uma requisição e em toda pilha de protocolos que essa requisição percorreu para ser processada. Os metadados X-Trace são localizados no interior da mensagem de aplicação e no interior dos pacotes de rede, mais especificamente, nos cabeçalhos dos pacotes. Esses metadados possuem dados relevantes de cada camada da pilha de protocolo e da requisição corrente. Os metadados são transportados nos campos de extensão, opção ou anotação dentro dos protocolos de rede, como, por exemplo, campo opção do protocolo IP, campo opção do protocolo TCP e dentro do cabeçalho HTTP.

X-Trace deve ser introduzido em um ambiente que segue alguns princípios para seu funcionamento. Um desses princípios é que os metadados devem percorrer o mesmo caminho que os dados para que, dessa forma, os caminhos causais construídos sejam confiáveis e precisos, pois assim, é possível capturar qual foi o caminho percorrido pelas requisições. Cada elemento instrumentado, seja aplicação ou protocolo, ao receber qualquer mensagem com metadado, deve gerar e enviar relatórios a um gerente central. Os relatórios contêm informações sobre o elemento que observou o tráfego de

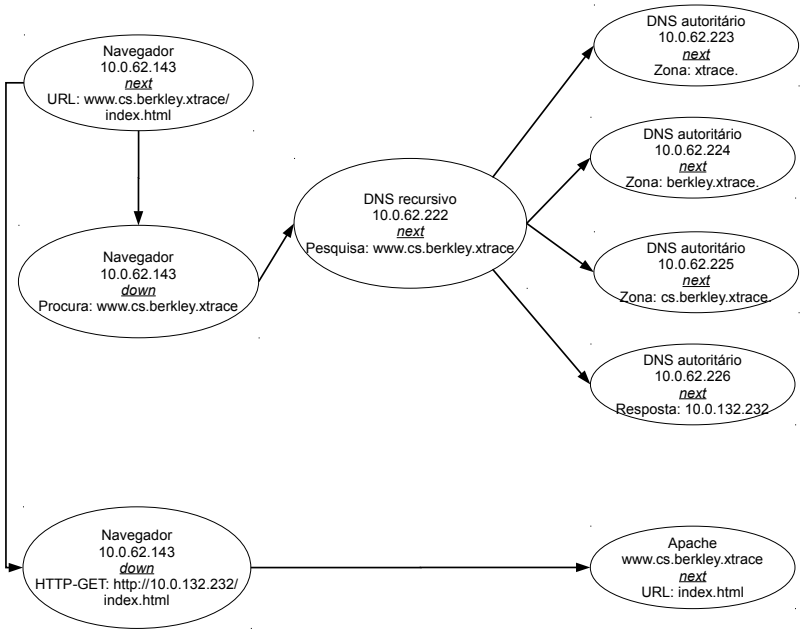


Figura 4.2: Árvore de tarefa de uma simples requisição *Web* com resolução de nomes utilizando DNS [35].

metadados. As informações encapsuladas nos relatórios, como carimbo de tempo, identificador e ligação entre os elementos, auxiliam a construção dos caminhos causais.

A adoção de X-Trace em uma rede envolve três passos: adicionar os metadados nas mensagens trocadas entre aplicações e entre as camadas de rede, adicionar lógica ao propagar os metadados seguindo os caminhos de dados das requisições e integrar os relatórios enviados pelos elementos instrumentados em árvores de tarefa.

A construção das árvores de tarefa é um processo semelhante ao da ferramenta Pinpoint, visto que identificadores únicos são trafegados no mesmo caminho dos dados, juntamente com as mensagens de aplicação. Dessa forma, mensagens de aplicação que possuem o mesmo identificador pertencem a uma mesma requisição. O grande diferencial de X-Trace é que a pilha de protocolo também é instrumentada, fazendo com que os protocolos também façam parte dos caminhos causais.

A Figura 4.2 ilustra um árvore de tarefa de uma requisição HTTP. O navegador faz uma requisição ao endereço `http://www.cs.berkley.xtrace/index.html`. Contudo, é

necessário resolver o nome do servidor em um endereço IP. Para isso, novas requisições são feitas para servidores de DNS para obtenção do endereço. Após várias requisições aos servidores de DNS em virtude do banco de dados de DNS ser distribuído, o navegador realiza a requisição para o servidor *Web* no endereço correto.

X-Trace foi avaliado em diversos serviços de rede [35], como: autenticação 802.1X, conteúdo *Web* distribuído (CoralCDN [10]) e seleção de réplicas baseada em DNS (OASIS [32]). As técnicas utilizadas para a instrumentação e os principais desafios encontrados durante as avaliações são demonstrados em [41].

Embora os resultados [35, 41] garantam a eficácia de X-Trace, sua utilização fica limitada apenas a um subconjunto de aplicações de rede em que seus códigos-fontes sejam disponíveis e que possam ser modificados. Além disso, a instrumentação é um processo que demanda tempo e é sucetível a erros, exigindo um conhecimento prévio e a documentação das aplicações.

4.2 Ferramentas Não Intrusivas

4.2.1 Sherlock

Sherlock [34] é o trabalho considerado mais completo para detecção automática de anomalias da literatura. Ele discute todos os passos necessários para obter detecções automáticas, desde a criação dos agentes até a localização de uma possível falha. Sherlock modela o problema de detecção de anomalias como uma Rede Bayesiana e realiza inferências probabilísticas para a detecção. Essa ferramenta utiliza o conceito de agente e gerente em sua estrutura. A Figura 4.3 apresenta uma visão geral de Sherlock.

Os agentes são instalados nas máquinas a serem monitoradas, possuem a função de analisar o tráfego de rede da máquina e coletam informações de cada serviço acessado. A partir dessas informações, é possível encontrar dependências entre serviços (baseado no tempo de coocorrência entre os acessos), todos os elementos de rede vinculados a um determinado serviço (baseado no aplicativo *traceroute*) e as observações dos serviços

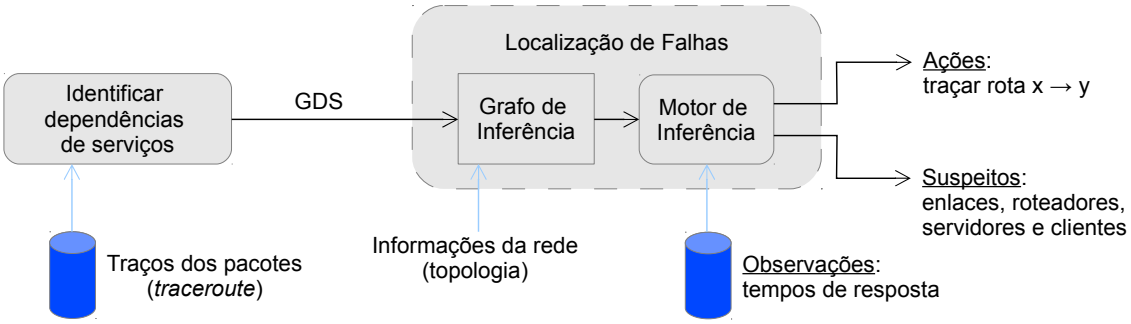


Figura 4.3: Visão geral de Sherlock [34].

(tempos de resposta). Com essas informações, os agentes produzem, localmente, um Grafo de Dependências de Serviços (GDS) e enviam o GDS para um gerente para construir um Grafo de Inferência (GI), que é uma caracterização de caminhos causais. A partir do GI e dos tempos de resposta dos serviços, o gerente realiza inferências para detectar anomalias.

GDS é um grafo que representa as dependências locais entre serviços. As dependências são determinadas a partir do tempo de coocorrência entre os acessos. Por exemplo, um *host* realiza uma requisição DNS para um servidor de nomes e obtém a resposta da requisição. A partir dessa resposta, o *host* faz uma requisição HTTP para um servidor *Web*. Se a janela de tempo entre a resposta DNS e a requisição HTTP for menor ou igual a 10 ms, então, há uma dependência entre esses acessos. A Figura 4.4 ilustra esse exemplo.

Claramente, inferência de Sherlock não é a melhor escolha para construir caminhos causais bem definidos, pois não leva em consideração o conteúdo das requisições, as aplicações, o sistema operacional, etc., apenas as conexões de rede e o tempo de coocorrência. Entretanto, os caminhos causais construídos por Sherlock simbolizados em GDSs e GIs são artefatos utilizados por essa ferramenta para detectar anomalias de rede.

Sherlock realiza a detecção de anomalias a partir das observações dos agentes sem prejudicar a utilização da rede e sem modificar qualquer aplicativo, utilizando somente informações sobre o tráfego de rede. Para realizar a detecção, Sherlock utiliza um algoritmo de inferência Bayesiana exponencial para inferir os possíveis causadores das

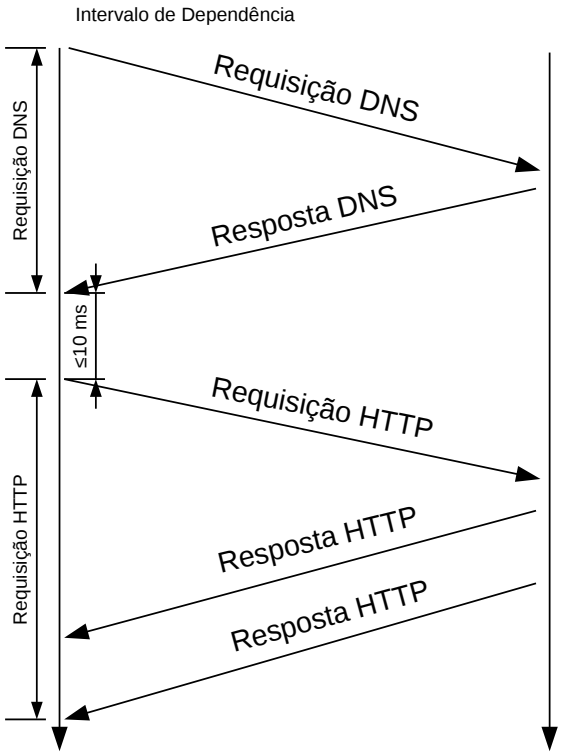


Figura 4.4: Dependência de acessos baseada no tempo de coocorrência entre a resposta DNS e a requisição HTTP.

anomalias. Contudo, devido a essa complexidade, a utilização de Sherlock não é escalável em centros de dados com grande quantidade de elementos. Entretanto, na literatura, diversas ferramentas de detecção de anomalias inspiradas em Sherlock foram desenvolvidas a fim de aperfeiçoar o desempenho de Sherlock, como Nemo [46] e Spotlight [42]. Nemo reduz o tempo gasto na inferência utilizando, principalmente, uma propriedade teórica de Redes Bayesianas e Spotlight comprime o GI, reduzindo a quantidade de nós e, consequentemente, reduzindo o tempo gasto na inferência.

4.2.2 BorderPatrol

BorderPatrol [37] é uma ferramenta que fornece caminhos causais sem modificar as aplicações e/ou *frameworks*, apenas modificando o fluxo de dados das aplicações. A modificação faz com que as mensagens de aplicação sejam repassadas para processadores de protocolo que extraem informações de processos, *threads* e mensagens de aplicação

para construir os caminhos.

BorderPatrol modifica o fluxo de dados para analisar as mensagens trocadas entre processos ou *threads* no exato momento que são trocadas. Essa análise é denominada observação ativa. A partir dessa análise, BorderPatrol utiliza os processadores de protocolo para obter informações das aplicações e construir os caminhos causais. Processadores de protocolo são abstrações de duas ideias combinadas: isolamento de eventos e testemunhas de mensagem.

O isolamento de eventos desagrega os eventos de entrada das aplicações para observar o comportamento da aplicação em uma visão "por evento". Dessa forma, qualquer atividade que a aplicação produza é resultado do único evento de entrada. As testemunhas de mensagem são identificadores encontrados nas mensagens trocadas pelos processos ou *threads*, normalmente, pares de pedido/resposta (*request/response*). Assim, se diferentes aplicações trocarem mensagens com um mesmo identificador (testemunha), então é possível realizar uma ligação entre as mensagens e, consequentemente, entre as aplicações.

Para que BorderPatrol possa construir corretamente os caminhos causais, a ferramenta supõe que as aplicações de rede são honestas, imediatas ou independentes. Uma aplicação é honesta quando os eventos de entrada e de saída utilizam o mesmo protocolo para realizar as comunicações. Dessa forma, alguns identificadores podem ser visíveis nas mensagens trocadas, facilitando as associações. Esses identificadores são denominados testemunhas.

As aplicações de rede nem sempre são honestas, neste caso, BorderPatrol utiliza as suposições imediatas e/ou independentes. Aplicações imediatas são aquelas que podem ser particionadas em fragmentos de processamento que não multiplexam requisições, fazendo com que os fragmentos processem, imediatamente, as entradas que lhes são fornecidas até o fim. Desse modo, BorderPatrol associa os eventos de entrada observando os eventos de saída do fragmento. Uma aplicação é independente quando fornece eventos de saída idênticos quando recebe eventos de entrada simultâneos ou sequenciais. Esta suposição não diz nada sobre as requisições, apenas sobre os eventos que as agregam.

A Figura 4.5 ilustra um exemplo de uma única aplicação de rede. O lado esquerdo

ilustra uma aplicação sem nenhuma suposição. Dessa forma, não é possível associar os eventos de entrada com os de saída, pois os módulos internos da aplicação são desconhecidos. O lado direito ilustra os módulos internos e, a partir das suposições imediata e independente, é possível realizar as associações dos eventos, visto que a suposição imediata diz que os fragmentos internos processam os eventos de entrada até o final e a suposição independente diz que os eventos de saída são idênticos, independentemente de os eventos de entrada terem sido fornecidos sequencial ou simultaneamente.

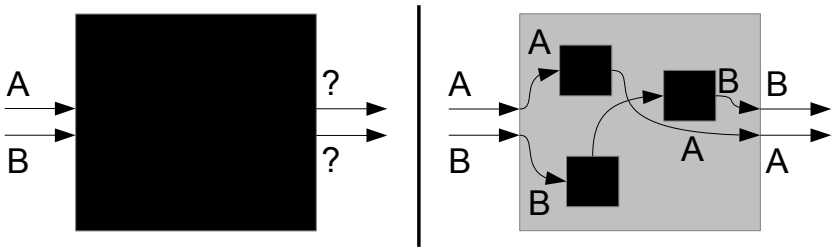


Figura 4.5: Exemplo de uma aplicação imediata e independente. Ela é imediata pois possui uma saída rotulada com A, mesmo rótulo do evento de entrada. A independência da aplicação é fornecida pois não processa a entrada A diferente se fosse fornecida simultaneamente com a entrada B [37].

A partir das suposições acima, BorderPatrol constrói os caminhos causais das requisições das aplicações sem modificá-las. Para isso, a ferramenta utiliza a técnica da observação ativa e os processadores de protocolo para compreender as mensagens de aplicação e obter as testemunhas de mensagem. Para realizar a observação ativa, BorderPatrol utiliza a técnica de interposição de biblioteca, onde os eventos de entrada e de saída das aplicações são tratados pelos processadores de protocolos antes de serem repassados para as aplicações de destino.

Na técnica de interposição de biblioteca, as aplicações carregam uma nova biblioteca dinâmica (*libbtrace*) antes de qualquer outra. As funções contidas em *libbtrace* são invólucros (*wrapper*) das funções originais da biblioteca `libc` [12]. As funções `open`, `socket`, `read` e `write` são algumas das funções implementadas em *libbtrace*. As funções de *libc* são invocadas como sub-rotinas das funções da *libbtrace* após repassar as mensagens de aplicação para o processador de protocolo.

Os processadores de protocolo, ao receber os eventos, buscam delimitar as mensagens de aplicação e salvar as testemunhas de cada mensagem. Quando não é possível extrair as testemunhas, as suposições imediatas e independentes são utilizadas para que os eventos de saída sejam originados pelo único evento de entrada do módulo da aplicação. Com isso, as ligações dos caminhos causais são facilmente realizadas. A Figura 4.6 ilustra um exemplo de caminho causal construído pela ferramenta.

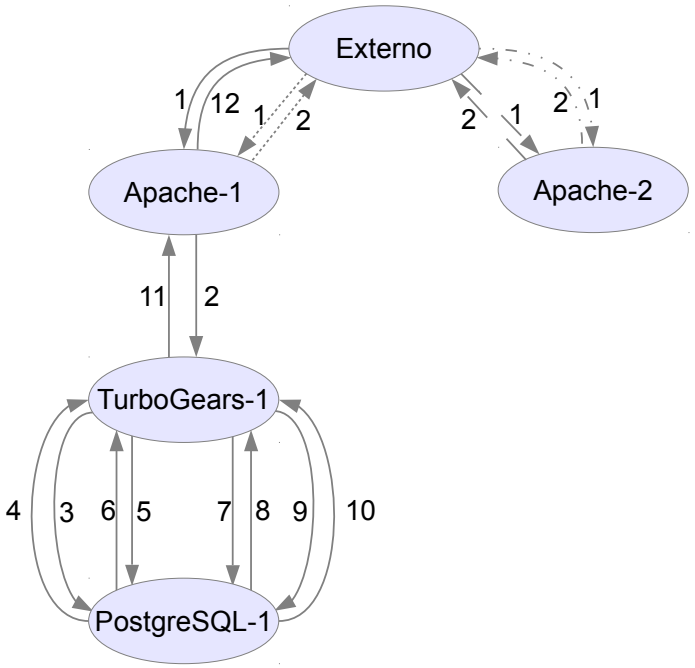


Figura 4.6: Exemplo de caminho causal construído por BorderPatrol de quatro requisições processadas por dois servidores `Apache`. Cada tipo de linha ilustra uma requisição diferente [37].

O caminho causal ilustrado na Figura 4.6 mostra que houve quatro requisições e que foram processadas por dois servidores `Apache`, sendo que três foram processadas diretamente pelos servidores *Web* e a outra teve a necessidade de conexões com os servidores `TurboGears` e `PostgreSQL` em seu processamento. Cada requisição é ilustrada por um tipo diferente de linha. Além disso, a Figura 4.6 mostra a sequência tomada por cada requisição ilustrada pelos números ao lado das linhas. Por exemplo, a requisição ilustrada pela linha pontilhada foi processada pelo servidor `Apache-1` em dois passos.

Apesar dos resultados apresentados em [37], as suposições honesta, imediata e independente nem sempre são verdadeiras, visto que, atualmente, há diversas técnicas de construção de aplicações distribuídas [26, 27]. Entretanto, como os processadores de protocolo são específicos de protocolo e não de aplicação, BorderPatrol obtém bons resultados em um conjunto significativo de aplicações.

4.3 Outros Trabalhos Relevantes

NSDMiner [44] é uma ferramenta automática de descoberta de dependências de serviços de rede por meio de observações do tráfego de rede. Ela determina a dependência entre os extremos de um fluxo observando o valor da razão entre a quantidade de vezes que o fluxo trafegou na rede e a quantidade de vezes que o serviço destino desse fluxo foi acessado. A ferramenta de detecção de anomalia NetMedic [39] constrói caminhos causais e detecta as anomalias de forma bastante detalhada devido às informações capturadas para definir os estados dos componentes de rede. NetMedic observa desde a utilização do processador até o tráfego de rede utilizado no processamento da requisição. A ferramenta Magpie [30] captura os recursos de máquina utilizados no processamento de requisições. Os administradores de rede devem fornecer esquemas de interação entre as aplicações para que Magpie associe os eventos capturados e construa os caminhos causais.

Outras ferramentas relacionadas ao assunto são: Macroscope [38], Project5 [29], Wap5 [33], eXpose [36] e Spotlight [42]. Elas contribuíram para formalizar o núcleo de S-Trace, pois colaboram com tema de construção de caminhos causais.

4.4 Considerações Finais

O processo de construção de caminhos causais é, normalmente, incorporado nas ferramentas de detecção de anomalias, pois os caminhos auxiliam a detecção por possuir todos os elementos utilizados no processamento de uma requisição. Essas ferramentas são classificadas em intrusivas e não intrusivas, o que determina se a ferramenta realiza

alguma modificação em aplicações, *frameworks* e sistemas operacionais ou não.

Este capítulo elenca quatro ferramentas que constroem caminhos causais para detectar as anomalias. As intrusivas possuem alta acurácia, porém necessitam de instrumentação, diferentemente das não intrusivas, em que nenhuma modificação é realizada. S-Trace explora os aspectos positivos das ferramentas intrusivas e não instrusivas, mais especificamente, de BorderPatrol e X-Trace.

Capítulo 5

Ferramenta de Construção S-Trace

S-Trace foi desenvolvida com base nas ferramentas X-Trace [35], BorderPatrol [37] e OFRewind [43] para construir e visualizar caminhos causais de granularidade fina. Ela explora informações provenientes de uma SDN [17] para identificar os componentes de hardware e software utilizados no processamento de uma requisição e para corrigir e aprimorar caminhos causais construídos somente com as informações dos extremos das conexões. S-Trace utiliza os conceitos de agente e gerente e seu fluxo geral está ilustrado na Figura 5.1.

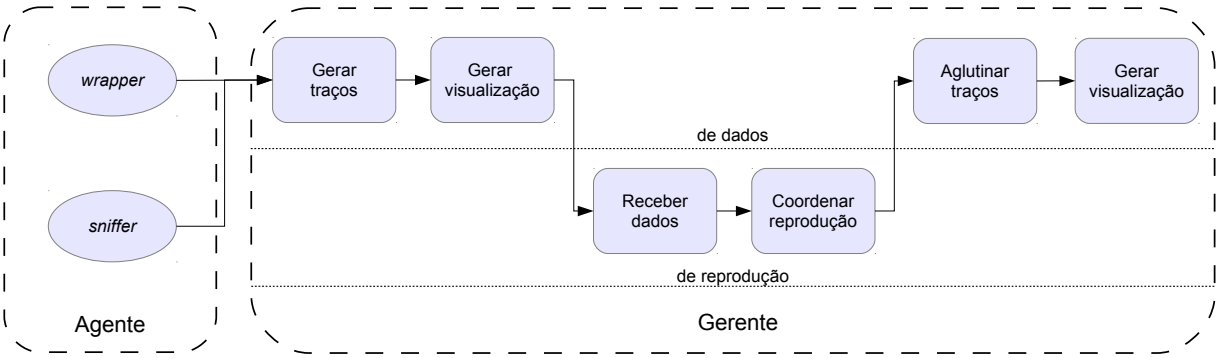


Figura 5.1: Fluxo geral da ferramenta S-Trace.

Os agentes, que são instalados em máquinas a serem monitoradas (estações de trabalho ou servidores), são divididos em dois componentes: *wrapper* e *sniffer*. O *wrapper* monitora

as chamadas de função utilizadas durante a execução das aplicações e o *sniffer* captura pacotes enviados e recebidos pelas estações monitoradas. Ambos enviam informações do monitoramento para o gerente de dados por meio de canais confiáveis e exclusivos, utilizando o protocolo TCP.

O gerente da ferramenta S-Trace recebe informações dos agentes e as consolida em caminhos causais. Para tanto, o gerente é dividido em gerente de dados e gerente de reprodução. O gerente de dados é responsável por receber informações enviadas pelos agentes, construir — em um estágio inicial — os caminhos causais e gerar um arquivo com os dados para a visualização dos caminhos. O gerente de reprodução é responsável por reproduzir o comportamento da rede original e fornecer informações ao gerente de dados para refinamento dos caminhos causais. A reprodução do comportamento da rede utiliza as mesmas técnicas de gravação e reprodução do tráfego de rede da ferramenta OFRewind.

As seções seguintes descrevem como todas as ações dos agentes e dos gerentes de S-Trace são realizadas e implementadas.

5.1 Agente

5.1.1 *wrapper*

O componente *wrapper* é encapsulado como uma biblioteca compartilhada (*libwrapper*) para monitorar chamadas de função que as aplicações realizam durante a execução, capturar informações relevantes desse monitoramento e enviar essas informações para o gerente de dados. Para isso, a técnica de interposição de biblioteca é utilizada. Essa técnica intercepta as chamadas para que funções da *libwrapper* sejam invocadas ao invés das funções da biblioteca original, sem ter que instrumentar as aplicações. A interposição de biblioteca faz com que a biblioteca compartilhada seja carregada antes de qualquer outra.

As funções da *libwrapper* são apenas invólucros das originais e o processamento

continua sendo realizado pela função original. Para isso, a função original é invocada no interior de cada uma das funções da biblioteca *libwrapper*. Com isso, o comportamento das funções e, consequentemente, das aplicações, não é alterado pelo componente *wrapper*. Todas as funções que são interceptadas pela *libwrapper* são definidas anteriormente e são mostradas na Seção 5.3, para que a biblioteca compartilhada *libwrapper* seja construída. A Figura 5.2 ilustra a criação da biblioteca e sua localização em uma aplicação.

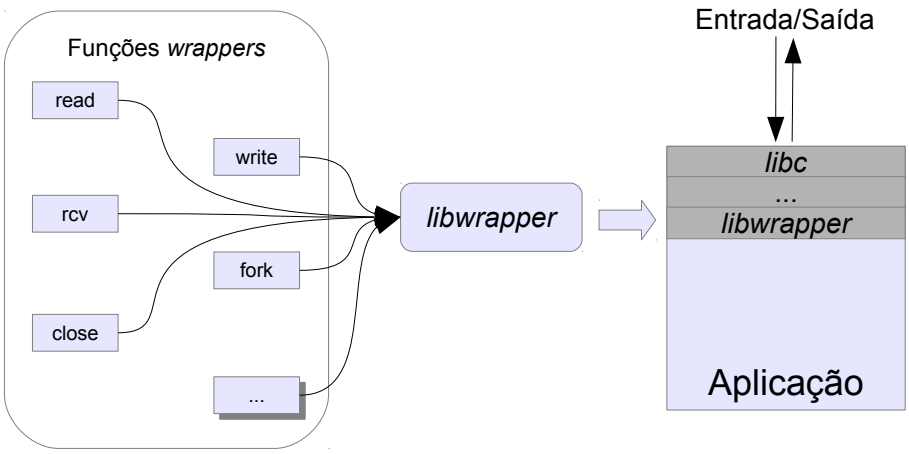


Figura 5.2: Criação da biblioteca *libwrapper* e sua localização em uma aplicação. Essa biblioteca é localizada na primeira camada superior da aplicação, a localização é fornecida pela técnica de interposição de biblioteca.

A Figura 5.3 ilustra um exemplo de como o fluxo de dados de uma aplicação é modificado quando a biblioteca *libwrapper* é utilizada pela aplicação. A aplicação invoca a função `read` para realizar a leitura de 512 bytes a partir do descritor de arquivo 20 e armazenar esses dados em *buffer*. Quando o componente *wrapper* é utilizado, essa invocação é encaminhada para função implementada pela *libwrapper*. A função `read` desta biblioteca invoca a função original da biblioteca *libc*, que por sua vez, realiza a leitura no descritor de arquivo. As linhas pontilhadas indicam como os dados são trafegados com destino à aplicação e a linha tracejada indica como seria o trafégo sem a utilização da biblioteca *libwrapper*. A linha contínua indica o fluxo de dados independente da utilização do componente *wrapper*.

As funções implementadas pelo componente *wrapper* possuem fluxo predefinido, que

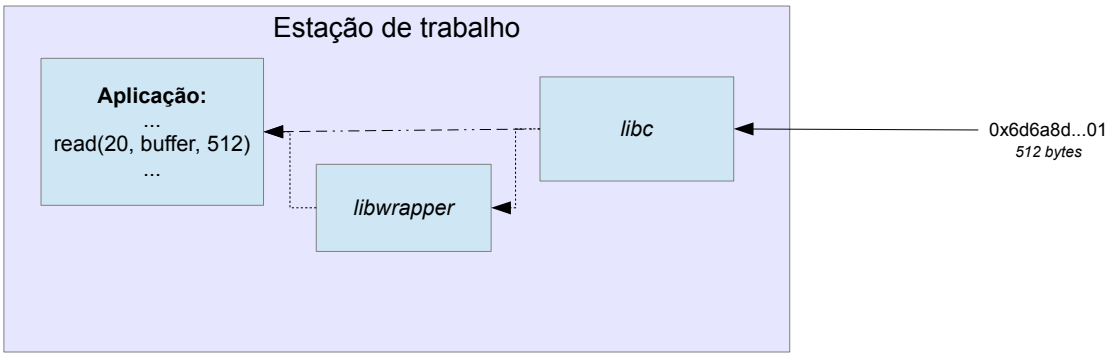


Figura 5.3: Exemplo do fluxo de dados de uma aplicação com e sem a biblioteca *libwrapper*. A linha contínua indica o fluxo de dados independente da utilização do componente *wrapper*. A linha tracejada indica o fluxo de dados sem a utilização da *libwrapper* e as linhas pontilhadas indicam o fluxo com ela.

é dividido em três fases. A primeira fase consiste em invocar a função original da biblioteca correta com os mesmos parâmetros, armazenar o valor de retorno da função original e retorná-lo pela função da *libwrapper*. Dessa forma, o comportamento da aplicação não é modificado. A segunda fase consiste em extrair informações referentes à aplicação que auxiliam a ferramenta S-Trace a construir os caminhos causais das requisições. Essas informações são ilustradas na Figura 5.4. A última fase consiste em enviar essas informações para o gerente de dados, que é descrito na Seção 5.2.1. A Função 5.1 ilustra trechos da implementação da função `read` da biblioteca *libwrapper*. As Linhas 5-6, 8-11 e 13 dessa função ilustram as três fases, respectivamente.

Função 5.1: Implementação do invólucro da função `read`.

```
1 ssize_t read(int fd, void* buf, size_t n) {
2   Metadata* data = (Metadata*) malloc(sizeof(Metadata));
3   static ssize_t (*read_real)(int, void*, size_t) = NULL;
4   ...
5   read_real = dlsym(RTLD_NEXT, "read");
6   ret_value = read_real(fd, buf, n);
7   ...
8   data->sec = htonl(get_timestamp().sec);
9   data->usec = htonl(get_timestamp().usec);
10  data->arg1 = htonl(ret_value);
11  data->pid = htons(syscall(SYS_getpid));
12  ...
13  send_datamanager(data);
14  return ret_value;
15 }
```

A implementação de uma função pela biblioteca *libwrapper* invoca a função original da biblioteca correta com os mesmos parâmetros recebidos para que o comportamento da aplicação não seja modificado. A função `dlsym` é utilizada para obter o endereço de memória da função correta, como ilustrado na Linha 5 da Função 5.1. Após obter o endereço, a função é invocada e seu valor de retorno é armazenado para que seja retornado ao final da função. As Linhas 6 e 14 da Função 5.1 ilustram esses fatos, respectivamente.

— 16 bits —	— 16 bits —
Operação	LWPID
PID	PPID
IP_{origem}	
$IP_{destino}$	
$PORTA_{origem}$	$PORTA_{destino}$
Argumento 1	
Argumento 2	
Argumento 3	
Argumento PP	
Tamanho da mensagem	
1º Caminho	
2º - 26º...	
27º Caminho	
Segundos	
Microsegundos	

Figura 5.4: Informações extraídas pelo *wrapper* e como são encapsuladas.

Após a invocação da função original, a função da *libwrapper* deve extrair e encapsular informações referentes à aplicação e à função. A Figura 5.4 ilustra essas informações e como são encapsuladas em uma estrutura. Porém, dependendo da função, alguns campos dessa estrutura podem não ser preenchidos, por exemplo, o campo `IP destino` na função `fork`. O campo `Operação` é utilizado para indicar qual foi a função invocada. Os campos `Argumento 1`, `Argumento 2` e `Argumento 3` são utilizados para armazenar informações da função, por exemplo, valor de retorno, descritor de arquivo, etc. O campo `Argumento PP` é utilizado pelos processadores de protocolo para indicar se a mensagem de aplicação possui atributos para apontar se essa mensagem é o final de uma conexão. Os campos `1º Caminho`, `2º Caminho`, `...`, `27º Caminho` são utilizados para armazenar o caminho absoluto de um arquivo. As informações extraídas e encapsuladas são enviadas para o gerente de dados por meio de um canal confiável, utilizando o protocolo TCP. As Linhas

8-*11* da Função 5.1 ilustram como as informações referentes à aplicação são extraídas e encapsuladas pela função `read` da *libwrapper*.

O processador de protocolo é uma função que interpreta as mensagens trocadas entre as aplicações e extrai informações sobre elas, por exemplo, identificando limitantes de mensagens de aplicação, atributos de protocolo, etc. O conhecimento necessário para o desenvolvimento de processadores é específico de protocolo e não de aplicação. O Algoritmo 5.1 ilustra um pequeno trecho do pseudo-código de um processador de protocolo HTTP simples. Os processadores de protocolo são semelhantes aos da ferramenta BorderPatrol descrita na Seção 4.2.2

A utilização do componente *wrapper* não modifica as aplicações e nem as funções originais. O componente modifica apenas o fluxo de dados das aplicações com um novo estágio adicionado ao caminho do fluxo. Esse estágio captura e envia informações sobre a aplicação para o gerente de dados e em seguida o fluxo prossegue normalmente. A adição desse novo estágio não afeta significativamente o desempenho da aplicação, pois são operações rápidas e curtas. O Capítulo 6 compara os tempos de execução de uma aplicação com e sem o *wrapper*.

Algoritmo 5.1: Processador de protocolo HTTP

Entrada: Mensagem de aplicação *buff*
Saída: Retorna *1* se a mensagem de aplicação indica o final da conexão

```
1  início
2      se "HTTP/1.0" ∈ buff então
3          retorna 1;
4      ...
5      se "\r\n\r\n" ∈ buff então
6          se "Connection: Keep-Alive" ∈ buff então
7              retorna 0;
8          senão
9              retorna 1;
10     retorna 0;
11 fim
```

5.1.2 *sniffer*

O componente *sniffer* monitora todos os pacotes enviados e recebidos pela estação de trabalho ou servidor e envia os cabeçalhos dos pacotes e informações relevantes sobre os cabeçalhos para o gerente de dados. Esse componente se comporta como *daemon*, ou seja, não há interação com os usuários. Apesar de capturar os pacotes, o *sniffer* não influencia no tráfego de rede, muito menos no comportamento das aplicações, pois esse componente realiza uma cópia dos pacotes para não alterar seu destino final. A Figura 5.5 ilustra um exemplo de uma estação de trabalho que recebe um pacote com destino à aplicação *App*, o qual é copiado para o componente *sniffer*.

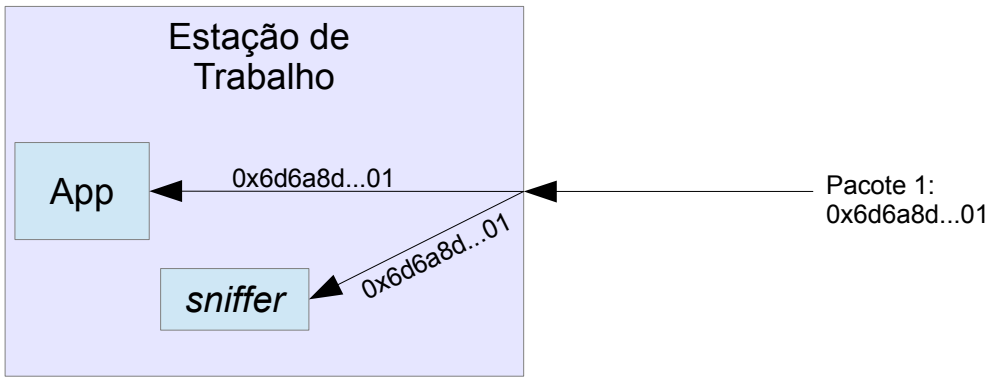


Figura 5.5: Exemplo do fluxo que um pacote percorre ao adentrar em uma estação de trabalho. O destino final do pacote é a aplicação *App*, contudo, uma cópia desse pacote é repassada ao componente *sniffer*.

Para realizar o monitoramento, o *sniffer* utiliza a API `pcap` (**p**acket **cap**ture) da biblioteca *libpcap* para realizar a captura dos pacotes. O *sniffer* permanece em um laço e, para cada pacote enviado ou recebido, uma função de *callback* é invocada pela *libpcap*. A Função 5.2 ilustra trechos da função de *callback* utilizada pelo *sniffer*.

Durante a execução da função de *callback*, as informações de tamanho dos cabeçalhos do pacote, o carimbo de tempo do momento em que o pacote trafegou pela estação de trabalho ou servidor e os cabeçalhos dos pacotes são encapsulados na estrutura de dados ilustrada na Figura 5.6. As Linhas 4 a 6 da Função 5.2 fazem o encapsulamento das

informações.

Função 5.2: Implementação da função de *callback* utilizada pelo *sniffer*.

```
1 void callback_function(unsigned char* args,
    ↪ const struct pcap_pkthdr* pkthdr,
    ↪ const unsigned char* packet) {
2   Metadata* data = (Metadata*) malloc(sizeof(Metadata));
3   ...
4   data->hlen = htonl(get_headers(packet).hlen);
5   data->sec = htonl(get_timestamp().sec);
6   data->usec = htonl(get_timestamp().usec);
7   ...
8   send_datamanager(data, packet, get_headers(packet).hlen);
9 }
```

O *sniffer* envia as informações encapsuladas por meio de um canal confiável para o gerente de dados, utilizando o protocolo TCP. Contudo, esse envio é realizado, normalmente, por meio da mesma interface de rede que é monitorada pelo componente *sniffer*. Dessa forma, há necessidade de um filtro especial no monitoramento dos pacotes para que os pacotes com destino e origem ao gerente de dados não sejam caputurados. Se esse filtro não fosse utilizado, haveria um laço infinito no monitoramento dos pacotes.

O modo de execução do *sniffer* não modifica o fluxo de dados, muito menos o comportamento das aplicações da estação de trabalho ou servidor. Esse componente apenas realiza uma cópia dos pacotes que a máquina envia ou recebe para extrair e enviar informações para o gerente de dados.

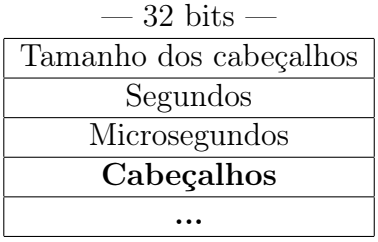


Figura 5.6: Informações extraídas pelo *sniffer* e como são encapsuladas.

5.2 Gerente

5.2.1 Gerente de Dados

O gerente de dados é um dos principais componentes da ferramenta S-Trace. Ele recebe os dados enviados pelos agentes, organiza e consolida esses dados, gera caminhos causais iniciais, produz uma visualização prévia dos caminhos, troca informações com o gerente de reprodução, aperfeiçoa os caminhos causais e, por fim, constrói os caminhos causais finais.

Para receber os dados enviados pelos agentes, o gerente de dados inicia duas conexões passivas, uma para os *wrappers* e outra para os *sniffers* em pontos distintos. A cada conexão estabelecida, uma nova *thread* é criada para processar essa conexão, recebendo os dados e adicionando-os em listas encadeadas ordenadas pelos carimbos de tempo. Portanto, inicialmente, o gerente de dados possui uma lista encadeada de nós *wrappers* e outra de nós *sniffers*.

Periodicamente, o gerente de dados inicia o processo de construção de caminhos causais a partir da lista encadeada de nós *wrappers*. Primeiramente, o processo busca por um nó que represente um início de conexão de rede, ou seja, um nó que represente uma função `accept` ou uma função `connect`. Se a busca não retornar um nó, o processo de construção é finalizado. Caso contrário, a partir do nó retornado, o processo tenta encontrar o último nó do caminho causal a ser construído, mais especificamente, algum nó que finalize a conexão representada pelo nó inicial, normalmente, um nó que representa a função `close`. Entretanto, o nó de finalização pode não ser encontrado. Nesse caso, a partir da porta de destino ou origem da conexão do nó inicial, o processo verifica se um processador de protocolo foi utilizado pelo componente *wrapper*. Em caso afirmativo, o campo `Argumento PP` de um nó *wrapper* sinaliza o nó final. A partir do nó inicial e do nó final do caminho causal, os nós intermediários desse intervalo entre os nós inicial e final são analisados, verificando se possuem alguma ligação com o caminho causal. Caso possuam ligações, esses nós são adicionados ao caminho. O Algoritmo 5.2 apresenta o pseudo-código para esse processo.

Algoritmo 5.2: Função para construção de caminhos causais

```
Entrada: Lista L encadeada com os nós wrappers
Saída: Caminho C causal
1  início
2      C = ∅;
3      n_i = encontrar_início(L);
4      se n_i = NULL então
5          retorna NULL;
6      n_n = encontrar_fim(L, n_i);
7      C.adiciona(n_i);
8      C.adiciona(n_n);
9      n = n_i.próximo;
10     enquanto n ≠ n_n faça
11         se possui_ligação(C, n) então
12             C.adiciona(n);
13         n = n.próximo;
14     retorna C;
15 fim
```

A função que verifica se um nó n possui algum tipo de ligação com um caminho causal C é determinada da seguinte maneira. Primeiramente, o conjunto S de LWPID (*Lightweight Process ID*) baseado nos nós de C é definido. Caso o valor do campo LWPID de n pertença a esse conjunto, n possui ligação com C. Caso contrário, a função de ligação verifica se n possui algum mecanismo de IPC com algum nó de C. Os tipos de IPC verificados são arquivos, *socket*, *pipe*, fila de mensagem, semáforos e memória compartilhada. Basicamente, a verificação desses mecanismos busca por identificadores comuns entre os nós. Por exemplo, se duas *threads* T_1 e T_2 do mesmo espaço de endereçamento realizam operações em um mesmo descritor de arquivo, então, há uma ligação entre as *threads*.

A Tabela 5.1 ilustra um exemplo de lista encadeada de nós *wrappers* manipulada pelo gerente de dados. O processo de construção de caminhos causais determina os nós 1 e 7 como nó inicial e nó final de um caminho causal, respectivamente. Portanto, apenas os nós que estão nesse intervalo são analisados. O conjunto S de LWPID de C consiste apenas em um único elemento $\{L_1\}$. Apesar dos nós 3 e 5 pertencerem ao processo P_1, eles não são adicionados ao caminho causal C, pois o valor da LWPID L_2 não pertence

Tabela 5.1: Exemplo da lista encadeada de nós *wrappers* com informações recebidas por um único processo, porém, *threads* diferentes.

Nó	**Tempo**	**LWP/PID**	**Operação**	**Desc. Arq.**	**IP/P**$_{origem}$	**IP/P**$_{destino}$
1	T_1	L_1/P_1	accept	20	(IP_1, P_1)	(IP_2, P_2)
2	T_2	L_1/P_1	read	20	(IP_1, P_1)	(IP_2, P_2)
3	T_3	L_2/P_1	connect	5	(IP_3, P_3)	(IP_4, P_4)
4	T_4	L_1/P_1	connect	22	(IP_2, P_5)	(IP_5, P_5)
5	T_5	L_2/P_1	write	5	(IP_3, P_3)	(IP_4, P_4)
6	T_6	L_1/P_1	write	22	(IP_2, P_5)	(IP_5, P_5)
7	T_7	L_1/P_1	close	20	–	–
...	...	...	...	...	...	...
i	T_i	L_1/P_1	shmget	–	–	–
$i+1$	T_{i+1}	L_1/P_1	fork	–	–	–
...	...	...	...	...	...	...

ao conjunto S e não possui nenhum tipo de ligação com os nós de C. Dessa forma, o caminho causal retornado na primeira iteração do processo de construção consiste nos nós $\{1, 2, 4, 6, 7\}$. Na segunda iteração do processo, os nós 3 e 5 farão parte de outro caminho causal, e assim por diante. Os nós i e $i+1$ que são executados pelo LWPID L_1 que pertence, de fato, à S, não são adicionados no caminho causal C, pois não estão contidos no intervalo do caminho, ou seja, entre os nós 1 e 7.

O processo de construção dos caminhos causais é invocado até que nenhum caminho causal seja retornado pela função. Todos os caminhos causais retornados pela função são adicionados em uma lista de caminhos causais. A partir dessa lista, é verificado se existe algum tipo de ligação entre os caminhos. A Figura 5.7 ilustra um exemplo de uma união entre os caminhos causais. A ligação é realizada pois os nós de início de conexão `connect` e `accept` possuem os mesmos endereços IP e números de porta, além disso, os carimbos de tempo indicam que os caminhos causais são compatíveis, ou seja, os caminhos podem ser compostos entre si.

Após as uniões entre os caminhos causais, o gerente de dados inicia o processo de rastreio entre as operações realizadas pelas aplicações (*wrapper*) e os cabeçalhos dos pacotes trafegados pelas estações de trabalho ou servidor (*sniffer*), ou seja, realiza a união entre os nós dos caminhos causais e os cabeçalhos dos pacotes. Essa operação é realizada por meio dos números de sequência utilizados pelo protocolo TCP.

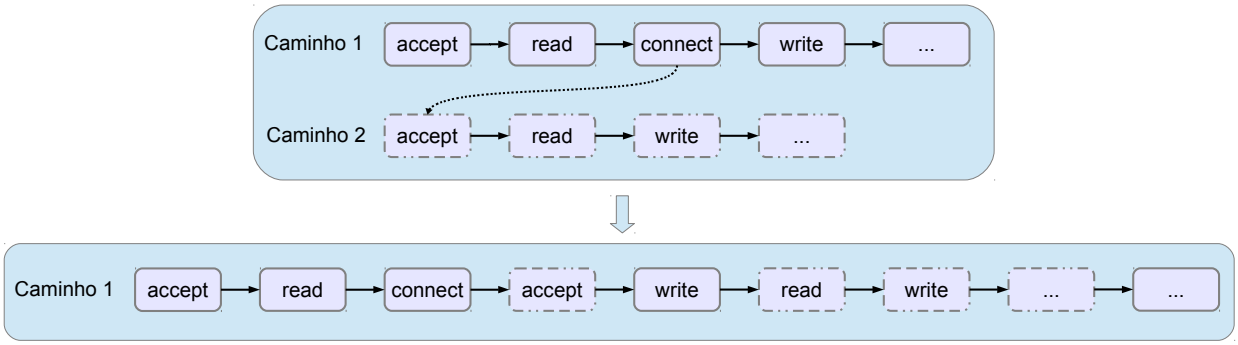


Figura 5.7: Exemplo de união entre caminhos causais. A linha tracejada indica que os dois nós possuem uma união, no caso, possuem os mesmos endereços IP e portas de origem e destino, indicando uma ligação.

A partir dos caminhos causais completos, ou seja, com os cabeçalhos associados, o gerente de dados inicia o processo de visualização utilizando os caminhos causais com os cabeçalhos associados. O gerente produz um único arquivo com extensão `.dot` (*graph description language*) contendo grafos que representam todos os caminhos causais construídos. Cada nó representa alguma chamada de função que a aplicação invocou. Caso essa chamada tenha resultado em pacotes de rede, novos nós são criados para representar esses pacotes. Após a construção do arquivo `.dot`, o gerente percorre cada caminho procurando por inícios de conexão que não foram unidos durante o processo de construção dos caminhos causais. Se existirem nós de conexão que não foram unidos, esses nós são informados ao gerente de reprodução, para que, posteriormente, caminhos causais sejam unidos, sem nenhum tipo de ligação aparente. O intuito dessa notificação é aprimorar os caminhos causais construídos pelo gerente de dados.

Em um primeiro momento, a notificação dos inícios de conexão para o gerente de reprodução não possui nenhum fundamento, pois se as informações fornecidas pelos agentes não forem suficientes, aparentemente não há ligação entre os caminhos. Entretanto, caso alguma técnica de modificação de campos do pacote seja utilizada pela rede, por exemplo, NAT, o gerente de dados e os agentes da ferramenta não saberiam dessa utilização. Porém, o gerente de reprodução possui essa informação que é fornecida por pela separação de planos de SDNs. A Figura 5.8 ilustra um exemplo de dois caminhos causais aparentemente sem nenhum tipo de ligação, porém, há uma regra em

um comutador que realiza a modificação dos endereços IP dos pacotes, fazendo com que os caminhos sejam ligados.

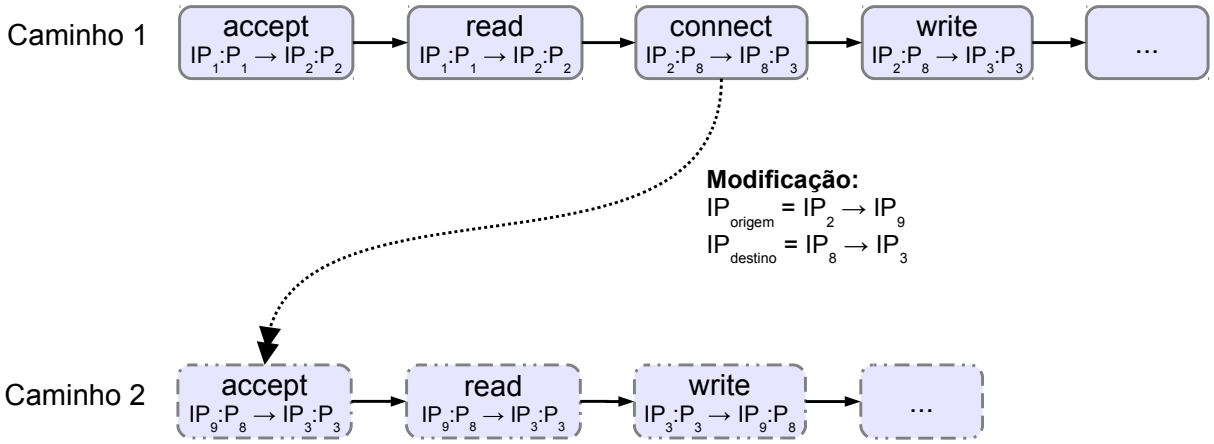


Figura 5.8: Exemplo de dois caminhos causais que não compartilham nenhuma informação de ligação entre os caminhos. Contudo, um comutador, que encaminha pacotes desses caminhos, realiza modificações nos endereços IP dos pacotes. Dessa forma, há, claramente, uma ligação entre os caminhos.

A simples utilização da técnica NAT faz com que um único caminho causal seja particionado, erroneamente, em duas requisições diferentes. Esse fato acontece pois não há nenhum tipo de união, nesse estágio, entre os caminhos. Entretanto, com a utilização do gerente de reprodução, esse evento é contornado, e os caminhos causais são unidos. Observando a Figura 5.8, sem a informação das modificações, os Caminhos 1 e 2 são totalmente distintos, o que é uma falsa dedução. As informações de modificações não são visíveis para os agentes e para o gerente de dados, apenas o gerente de reprodução possui essa informação, pois manipula as regras de fluxos instaladas nos comutadores de rede.

O gerente de reprodução, após receber os inícios de conexão, realiza diversos procedimentos (que são explicados detalhadamente na Seção 5.2.2) e envia ao gerente de dados informações de instalações de regras nos comutadores de rede referentes aos dados recebidos. Essas informações são utilizadas pelo gerente de dados para realizar uniões entre os caminhos causais que, aparentemente, não possuiam nenhum tipo de ligação. Dessa forma, o gerente de dados novamente invoca o processo de união entre os

caminhos causais juntamente com os dados recebidos pelo gerente de reprodução. Feito isso, o processo de visualização também é invocado novamente, gerando um novo arquivo `.dot` final, com os caminhos causais unidos e bem definidos.

Todos os estágios descritos são executados periodicamente pelo gerente de dados.

5.2.2 Gerente de Reprodução

O gerente de reprodução possui o objetivo de gravar o trafégo de rede de modo escalável, consistente e controlado, receber informações sobre os caminhos causais fornecidos pelo gerente de dados, coordenar a reprodução do tráfego gravado e informar ao gerente de dados as regras de fluxo que influenciam nos caminhos causais e que foram instaladas nos comutadores OpenFlow. O gerente de reprodução é composto da ferramenta OFRewind [43] e de dois novos módulos. Sua arquitetura é ilustrada na Figura 5.9.

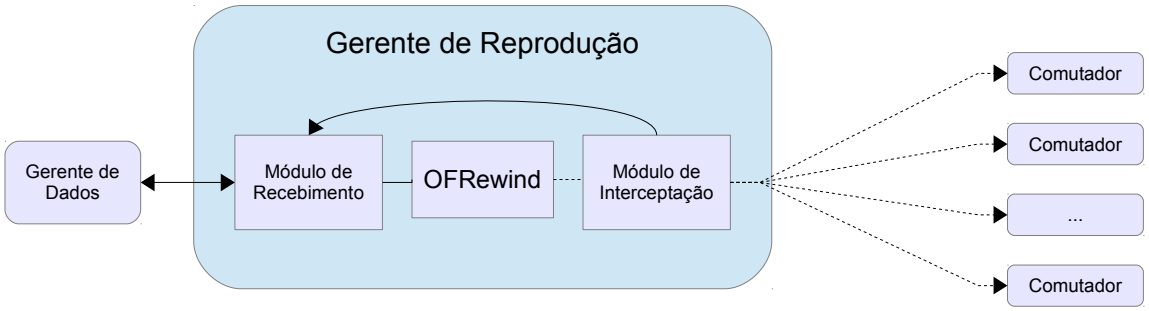


Figura 5.9: Visão geral dos componentes do gerente de reprodução e suas ligações. O módulo de recebimento possui ligação direta com o gerente de dados e com o módulo de interceptação. Os comutadores possuem canais com o componente OFRewind. Contudo, esses canais são interceptados pelo módulo de interceptação. Ambos módulos possuem ligações com o componente OFRewind.

A ferramenta OFRewind é o núcleo central do gerente de reprodução que consolida as técnicas de gravação e reprodução de tráfego de rede em um ambiente programável utilizando o protocolo OpenFlow. O gerente de reprodução explora essas técnicas para notificar qual foi o comportamento da rede durante o processo de gravação, ou seja, quais foram as ações tomadas pelos comutadores de rede. OFRewind foi sucintamente descrita na Seção 3.3.

Os dois módulos agregados no núcleo central do gerente de reprodução são responsáveis por receber e enviar mensagens para o gerente de dados. Essas mensagens refinam os caminhos causais gerados pelo gerente de dados. O módulo responsável por receber mensagens é instanciado simultaneamente com o gerente de reprodução, mais especificamente, uma nova *thread* é criada para gerenciar esse módulo. O módulo responsável por interceptar regras é anexado ao módulo de inserção de regras do modo de reprodução da ferramenta OFRewind.

A *thread* que gerencia o módulo de recebimento inicia uma conexão passiva para receber as informações do gerente de dados. Para cada conexão estabelecida, uma nova *thread* é criada para receber os dados e adicioná-los a uma lista encadeada. A estrutura dos dados recebidos é idêntica à estrutura utilizada pelo componente *wrapper* dos agentes, ilustrada na Figura 5.4. Portanto, essa lista encadeada é composta de nós *wrappers*.

O módulo de interceptação, que envia informação para o gerente de dados, está incorporado ao módulo de reprodução da ferramenta OFRewind, mais especificamente, na parte de instalação de regras nos comutadores. A atuação desse módulo não modifica o comportamento do OFRewind, apenas modificando o fluxo original da regra para extrair informações relevantes da regra a ser instalada. A estrutura das informações enviadas é idêntica à estrutura utilizada pelo módulo de recebimento. O Algoritmo 5.3 e a Figura 5.10 ilustram um pseudo-código da implementação desse componente e o fluxo de dados modificado, respectivamente.

Algoritmo 5.3: Função do módulo de envio de informações

```
   Entrada: Regra R de fluxo e lista L ligada
1  início
2      info = extrai_campos_do_cabeçalho(R);
3      nó_wrapper = encapsula(info);
4      se nó_wrapper ∈ L então
5          ações = extrai_ações(R);
6          nó_wrapper = encapusula(ações);
7          envia_para_gerente_de_dados(nó_wrapper);
8  fim
```

A Linha 2 do Algoritmo 5.3 extrai informações da regra de fluxo a ser instalada em um comutador — por exemplo, qual é o IP de origem, IP de destino, porta de origem, etc. —

e a Linha 3 cria um nó do tipo *wrapper* com essas informações. Se esse nó pertence à lista encadeada, então a regra a ser instalada em um comutador remete diretamente a algum início de conexão indicado pelo gerente de dados. Portanto, é necessário compreender quais são as ações associadas a essa regra que refletem na construção do caminho causal. Para isso, o campo `ações` da regra é encapsulado em um nó *wrapper*, que é enviado para o gerente de dados. Essas ações são explicitadas nas Linhas 5, 6 e 7.

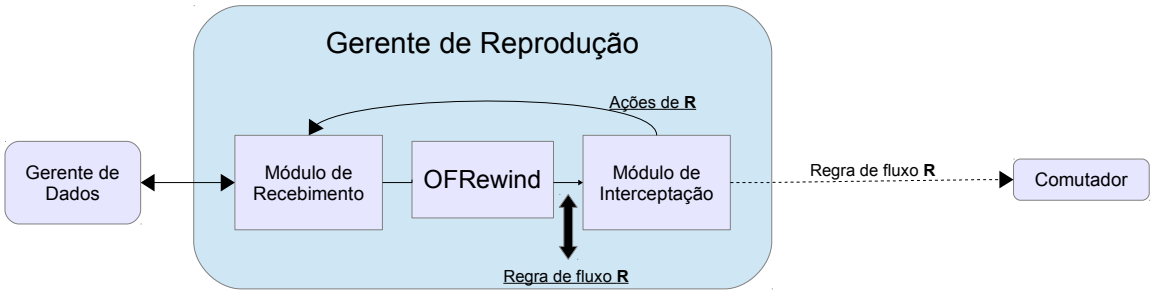


Figura 5.10: Fluxo de dados do gerente de reprodução. O componente do OFRewind fornece a regra **R** para ser instalada em um comutador. O módulo de interceptação captura a regra **R**, verifica se possui alguma ligação com os dados recebidos pelo módulo de recebimento, extrai as ações de **R** e envia para o gerente de recebimento que é enviado ao gerente de dados. Posterior a essa interceptação, a regra **R** é encaminhada, sem nenhuma modificação, para o comutador.

O gerente de reprodução é um componente fundamental para que o gerente de dados produza caminhos causais confiáveis e bem definidos, pois a reprodução do comportamento da rede revela relacionamentos que não podem ser capturados nos extremos de uma conexão. Essa contribuição é realizada, principalmente, pela captura das ações das regras de fluxos instaladas nos comutadores OpenFlow, fazendo que modificações nos pacotes de redes sejam observadas e notificadas para o gerente de dados por meio da interceptação das mensagens do plano de controle. Além disso, como todas as regras são instaladas em um momento pós-gravação, o comportamento do tráfego de rede não é alterado. Essa é uma característica importante da ferramenta OFRewind que é mantida pelo gerente de reprodução.

5.3 Aspectos de Implementação

A ferramenta S-Trace, composta de agentes e gerentes, foi implementada na linguagem `C` em um ambiente Linux. As bibliotecas *libpcap*, *libpthread*, *libdl* e as demais nativas do ambiente Linux foram utilizadas nessa implementação. A linguagem `C` foi escolhida por ser comum no desenvolvimento de sistemas operacionais, fornecendo acesso de baixo nível à memória e por possuir tratamento de operações de baixo nível, além de possuir um ótimo desempenho.

Tabela 5.2: Funções interceptadas pelo módulo *wrapper*.

`read (socket)`	`read (pipe)`	`recv (socket)`	`recv (pipe)`
`write (socket)`	`write (pipe)`	`send (socket)`	`send (pipe)`
`socket`	`bind (AF_UNIX)`	`bind (AF_INET)`	`listen`
`accept`	`connect`	`close`	`fopen`
`fclose`	`dup`	`dup2`	`fork`
`msgget`	`msgsnd`	`msgrcv`	`shmget`
`shmat`	`semget`	`semop`	`pipe`
`pipe2`	`mkfifo`	`readv (socket)`	`readv (pipe)`
`writev (socket)`	`writev (pipe)`	`sendfile`	

5.3.1 *wrapper*

O componente *wrapper* é responsável por interceptar as chamadas de certas funções para obter informações referentes à aplicação e à função. Para isso, esse componente é encapsulado como uma biblioteca compartilhada (*libwrapper*) que é carregada dinamicamente na execução das aplicações. Dessa forma, os dados da *libwrapper* não são copiados para o executável da aplicação. Com isso, diversas aplicações podem utilizar essa biblioteca ao mesmo tempo. Todas as funções interceptadas pelo componente *wrapper* estão ilustradas na Tabela 5.2. Os códigos-fonte de cada uma das funções implementadas pela *libwrapper* estão listados no Apêndice A.

A criação da biblioteca *libwrapper* é realizada por meio do compilador *gcc* [1] utilizando as flags `-shared -ldl -fPIC` para indicar que a saída é um objeto compartilhado, que utiliza a biblioteca `libdl` e que a implementação é independente de

posição, respectivamente. A execução do compilador para criação da *libwrapper* é apresentada abaixo:

```
gcc wrapper.c -o libwrapper.so -shared -fPIC -ldl
```

As aplicações devem carregar a biblioteca compartilhada antes de qualquer outra para que as chamadas de função sejam interceptadas corretamente. Para isso, a variável de ambiente `LD_PRELOAD` deve sempre conter o caminho da *libwrapper*, fazendo com que essa biblioteca seja carregada no início da aplicação. Por exemplo, a execução da aplicação *foo* com essa biblioteca carregada dinamicamente é demonstrada abaixo:

```
LD_PRELOAD=/var/lib/libwrapper.so ./foo -p 8080
```

A implementação de todas as funções *wrappers* deve, obrigatoriamente, invocar a função original sem modificações dos parâmetros. Para isso, a função `dlsym` (fornecida pela biblioteca *libdl*) é executada para obter o endereço de memória da função correta. `dlsym` necessita de dois argumentos: o primeiro é o manipulador de biblioteca e o outro é o nome do símbolo a ser procurado, no caso, o nome da função. Contudo, como a aplicação, a priori, já deveria ter carregado a biblioteca para a invocação da função, o manipulador de biblioteca é determinado pelo pseudo-manipulador **RTLD_NEXT** fazendo com que a função `dlsym` procure a próxima ocorrência da função nas bibliotecas posteriores a *libwrapper*.

Após invocar a função original, o componente *wrapper* extrai informações e as encapsula na estrutura ilustrada na Figura 5.4. O componente *wrapper* envia a estrutura preenchida de acordo com as funções invocadas para o gerente de dados por meio de um canal confiável, utilizando o protocolo TCP.

5.3.2 *sniffer*

O *sniffer*, componente do agente da ferramenta S-Trace, utiliza a biblioteca *libpcap* para monitorar todos os pacotes enviados e recebidos pela estação de trabalho ou servidor. Para isso, as interfaces são observadas utilizando a função `pcap_open_live` e todas as interfaces são monitoradas, pois o componente não tem o conhecimento prévio sobre as ligações de cada interface.

Algoritmo 5.4: Componente *sniffer*

```
1  início
2      ...
3      FILTER = "not host IP_GERENTE_DADOS";
4      manipulador = pcap_open_live("any", SNAP_LEN, PROMISC, TIMEOUT,
       errbuf);
5      ...
6      pcap_compile(manipulador, &fp, FILTER, OPTIMIZE, net);
7      pcap_setfilter(manipulador, &fp);
8      pcap_loop(manipulador, COUNT, função_callback, retvalue);
9      pcap_close(manipulador);
10 fim
```

A função `pcap_loop` é utilizada pelo *sniffer* para que qualquer pacote trafegado pela máquina seja interceptado por uma função de *callback*. Dessa forma, o componente fica em um laço infinito monitorando todos os pacotes. Note que a interceptação não altera o caminho original do pacote. Ela apenas realiza uma cópia para o processamento sem alterar o comportamento de nenhuma aplicação. A função de *callback*, representada pela Função 5.2, extrai o carimbo de tempo do momento em que o pacote trafegou pela máquina e os cabeçalhos desse pacote para que sejam enviados para o gerente de dados. Contudo, como o componente monitora todas as interfaces, os pacotes com o destino ao gerente também seriam monitorados. Para que os pacotes de redes com origem e destino ao gerente de dados não sejam processados pela função de *callback*, as funções `pcap_compile` e `pcap_setfilter` são utilizadas. O Algoritmo 5.4 mostra trechos da implementação do componente.

5.3.3 Gerente de Reprodução

O gerente de reprodução também foi desenvolvido na linguagem `C`, mesma linguagem utilizada pelo protótipo da ferramenta OFRewind disponibilizado em [18]. Por ser um protótipo, o código-fonte possui diversos pontos sem manutenção e suporte. Dessa forma, o protótipo foi modificado para ser utilizado pelo gerente de reprodução no ambiente de virtualização Mininet [40]. Uma das modificações realizadas no código-fonte foi a troca de mensagens simétricas, mais especificamente, as mensagens **Echo** que indicam a latência, largura de banda e/ou a conectividade da conexão entre o controlador e os comutadores. Os *switches* do ambiente Mininet enviam mensagens do tipo **Echo Request** para os controladores, aguardando respostas do tipo **Echo Reply**. Entretanto, a implementação do OFRewind fazia com que o controlador enviasse as requisições **Echo** aguardando resposta dos comutadores. Dessa forma, a conexão entre esses elementos era perdida, não sendo possível a troca de mensagens entre os comutadores e o controlador. Portanto, a ferramenta OFRewind foi modificada para receber mensagens **Echo Request** e responder com **Echo Reply**, permanecendo com as conexões ativas.

Os dois novos módulos do gerente de reprodução utilizam o modelo cliente-servidor em suas implementações, sendo que uma nova *thread* é criada para cada nova requisição do cliente. Essa técnica foi escolhida, pois, de acordo com [26], é uma técnica rápida, o servidor não possui uma carga de processamento pesada e o sistema operacional fornece mecanismos de *threads*.

Os dados recebidos pelo módulo de recebimento são adicionados em um lista encadeada e representam os inícios de conexão que não foram combinados na construção dos caminhos causais. Portanto, somente nós que representam as funções `accept` ou `connect` estão nessa lista.

O módulo que intercepta a instalação de regras de fluxo utiliza essa lista encadeada para verificar se alguma regra a ser instalada nos comutadores possui alguma ligação com os nós da lista. Somente o tipo `OFPT_FLOW_MOD` de mensagem das regras de fluxo é observado. Se não possuir, não há interceptação e a execução prossegue normalmente.

Entretanto, se tiver alguma ligação, a regra pode influenciar a construção dos caminho causais, ou seja, os comutadores podem alterar campos dos pacotes, dificultando a ligação dos nós da lista e, consequentemente, a construção dos caminhos causais. A ligação é baseada nos campos endereços IP de origem e destino e portas de origem e destino das regras de fluxos. Se os campos possuem ligações, então a regra de fluxo é expandida para obter as ações da regra. Mais especificamente, os tipos `OFPAT_SET_NW_SRC, OFPAT_SET_NW_DST, OFPAT_SET_TP_SRC` e `OFPAT_SET_TP_DST` [15] são observados para verificar se os comutadores modificam os campos de endereço IP ou portas dos pacotes, respectivamente.

O módulo de interceptação, ao observar uma regra R que possui ligação com algum nó n da lista encadeada, produz um novo nó n' em que os campos endereço IP e porta de origem e destino são preenchidos com os valores que pertencem às ações de R. Além disso, os campos `Segundos` e `Microsegundos` do nó n' são preenchidos com os mesmos valores que o nó n. O nó n' preenchido é enviado para o gerente de dados para aperfeiçoar os caminhos causais produzidos pela ferramenta S-Trace. Quando o gerente de dados recebe esse nó, ele é tratado como um nó *wrapper* normal, ou seja, é adicionado à lista encadeada e o processo de construção de caminhos causais é invocado novamente.

5.3.4 Gerente de Dados

O principal componente da ferramenta S-Trace é o gerente de dados, que consolida informações dos *wrappers* e *sniffers* e constrói os caminhos causais. Esse componente inicia três conexões passivas para receber os dados dos agentes e do gerente de reprodução. Todas essas conexões utilizam o modelo cliente-servidor com a mesma técnica utilizada pelo gerente de reprodução. As informações recebidas são adicionadas em três diferentes listas: uma para os *wrappers*, uma para os *sniffers* e outra para o gerente de reprodução.

O gerente de dados possui três módulos que são responsáveis pelas conexões passivas para o recebimento de dados. As mensagens trocadas pelos agentes e gerente são do tipo assíncronas e diversas mensagens podem ser encaminhadas a qualquer momento,

inclusive durante o processo de construção dos caminhos causais. Para que a construção dos caminhos não interfira no recebimento de dados, o gerente de dados possui dois ponteiros extras: um para a lista dos *wrappers* e outro para os *sniffers*. Quando o processo é iniciado, há uma troca de ponteiros para que as listas não sejam modificadas durante o processo de construção e para que os recebimentos não sejam afetados.

Todas as listas utilizadas pelo gerente de dados são implementadas como listas encadeadas e os ponteiros de cabeça e de cauda são armazenados. O ponteiro de cauda é utilizado para facilitar a adição de novos nós, visto que as listas são ordenadas de forma não decrescente pelo carimbo de tempo dos nós.

O processo de construção dos caminhos causais é realizado a cada cinco minutos. A ação inicial desse processo é retornar uma lista encadeada de caminhos causais conforme descrito na Seção 5.2.1. Após obter a lista encadeada de caminhos causais, cada início de conexão que não foi combinado é enviado para o gerente de reprodução. Mais especificamente, o nó correspondente é enviado. A estrutura desse nó é ilustrada na Figura 5.4.

A partir dos caminhos causais definidos, o gerente de dados utiliza os dados enviados pelos *sniffers* para combinar as operações de E/S dos caminhos causais com os cabeçalhos dos pacotes. Essa combinação é dividida em duas partes, a primeira coleta todas operações de entrada e todos os cabeçalhos de entrada e a outra coleta os dados de saída. Com isso, a operação de combinação é realizada por meio dos números de sequência dos pacotes TCP. Essa operação é realizada pois os nós *wrappers* estão ordenados pelo carimbo de tempo, garantindo a ordem de invocação das funções. Entretanto, o carimbo de tempo dos nós *sniffers* não é um forte parâmetro para realizar essa combinação, pois não há garantia da ordem de saída ou entrada dos pacotes por meio do tempo. Os números de sequência são utilizados para realizar as combinações dos nós *wrappers* com os nós *sniffers*, ou seja, as operações de E/S com os cabeçalhos dos pacotes.

O gerente de dados, após realizar a combinação dos nós *wrappers* com os cabeçalhos fornecidos pelos *sniffers*, inicia o módulo de visualização. Esse módulo produz um único arquivo `.dot` (*graph description language*) contendo todos os caminhos causais

representados por grafos direcionados. O Capítulo 6 ilustra alguns exemplos dos grafos produzidos pelo gerente de dados.

Os dados recebidos pelo módulo de recebimento do gerente de dados utiliza a mesma estrutura do componente *wrapper*. Contudo, como já foi descrito pelo gerente de reprodução, o modo de preenchimento dos campos da estrutura é diferenciado. Apenas os campos IP_{origem}, $IP_{destino}$, $PORTA_{origem}$, $PORTA_{destino}$, `Argumento 3`, `Segundos` e `Microsegundos` são utilizados. Quando esse módulo recebe um dado encapsulado na estrutura com os campos `Segundos` e `Microsegundos` iguais a 0, isso indica o fim da reprodução pelo gerente de reprodução, dessa forma, o gerente de dados inicia, novamente, o processo de construção de caminhos causais e de visualização. Por outro lado, quando os campos `Segundos` e `Microsegundos` são diferentes de zero, isso indica que durante o tráfego real algum comutador realizou modificações nos pacotes de algum fluxo. Essas modificações são observadas nos campos IP_{origem}, $IP_{destino}$, $PORTA_{origem}$ e $PORTA_{destino}$ desse nó.

Após receber todas as informações do gerente de reprodução, o processo de construção de caminhos causais é invocado. Quando há algum tipo de combinação entre os caminhos causais, esse processo verifica se algum dos nós combinados é um nó recebido pelo gerente de reprodução, ou seja, um nó especial. Caso o campo `Argumento 3` possua o valor `(unsigned int) -1`, o nó é especial. Quando isso acontece, o processo deve procurar pelo nó original para fazer a combinação correta. Para isso, ele procura pelo nó original que possui o mesmo valor dos campos `Segundos` e `Microsegundos` do nó especial.

5.4 Considerações Finais

Este capítulo descreve S-Trace, uma nova ferramenta de construção de caminhos causais de requisições sob o paradigma de Redes Definidas por Software (SDNs). S-Trace explora pontos positivos da construção de caminhos causais de ferramentas de detecção de anomalias intrusivas e não intrusivas. Além disso, ela utiliza o protocolo OpenFlow para realizar as técnicas de gravação e reprodução do tráfego de rede para

que assim possa construir os caminhos causais de maneira eficiente e eficaz. O capítulo também descreve os principais componentes de S-Trace e os aspectos de implementação da ferramenta.

Os caminhos causais construídos por S-Trace auxiliam a compreender o comportamento de aplicações de rede sem ter acesso aos componentes internos das aplicações. Além disso, os caminhos facilitam detecções de anomalias realizadas por ferramentas especializadas, pois eles compreendem todos os eventos realizados pela aplicação para processar a requisição de rede. S-Trace constrói caminhos causais com precisão de ferramentas intrusivas sem modificar aplicações, *frameworks* ou sistema operacional assim como as ferramentas não intrusivas. Dessa forma, S-Trace se torna uma interessante ferramenta de construção de caminhos causais em centros de dados modernos com diversas aplicações e diferentes dispositivos de rede.

Capítulo 6

Avaliação e Uso de S-Trace

S-Trace foi avaliada utilizando a ferramenta Mininet [40] que possibilita a criação de diferentes redes virtuais em um único computador, além de fornecer a rápida prototipação de diferentes redes virtuais baseadas no protocolo OpenFlow [21]. Mininet é executada em uma máquina virtual e sua imagem, disponível em [11], pode ser instanciada em um gerenciador de máquinas virtuais, como VirtualBox [13].

A avaliação da ferramenta S-Trace possui o intuito de mostrar que os caminhos causais construídos refletem de maneira confiável quais foram os eventos utilizados pelas *threads* ao processar uma requisição. Foram utilizadas duas aplicações para a avaliação e algumas falhas foram produzidas para demonstrar que os caminhos causais construídos são capazes de auxiliar a detecção dessas falhas, além de expor o comportamento das aplicações.

As próximas seções descrevem as aplicações utilizadas, as falhas injetadas e como S-Trace lida com elas e os resultados obtidos por S-Trace.

6.1 Avaliação

A avaliação da ferramenta foi realizada sob uma única topologia de rede ilustrada na Figura 6.1. O cenário possui quatro comutadores OpenFlow e 13 *hosts*. Cada *host* executa uma aplicação específica, que é explicada nas subseções seguintes.

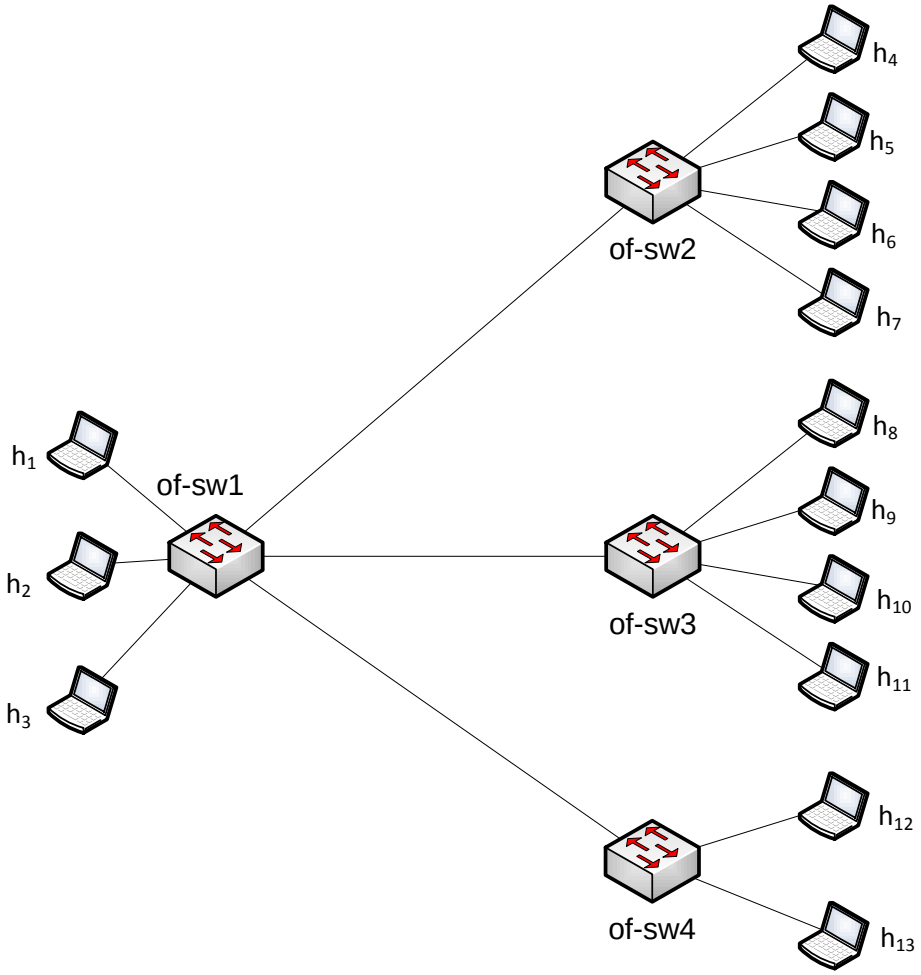


Figura 6.1: Topologia com quatro comutadores e 13 *hosts* utilizada na avaliação da ferramenta S-Trace.

Mininet disponibiliza um controlador padrão que utiliza a Versão 1.0.0 do protocolo OpenFlow [15]. Esse controlador é instanciado na máquina virtualizada juntamente com Mininet, dessa forma, o controlador não está presente na rede produzida por Mininet.

6.1.1 Aplicação n-Tier

A aplicação n-Tier foi desenvolvida com intuito de simular o comportamento de aplicações que utilizam múltiplas camadas, fazendo com que a avaliação dessa aplicação cubra grande parte das aplicações distribuídas usuais. A aplicação foi instrumentada para que forneça todos os eventos realizados pelas *threads* no processamento das requisições. Com isso, é possível comparar os caminhos causais construídos por S-Trace e os eventos fornecidos pela instrumentação. Cada instância de n-Tier é denominada réplica e possui uma *thread* principal, além de um *pool* de *threads*. A Figura 6.2 ilustra a visão geral da aplicação n-Tier.

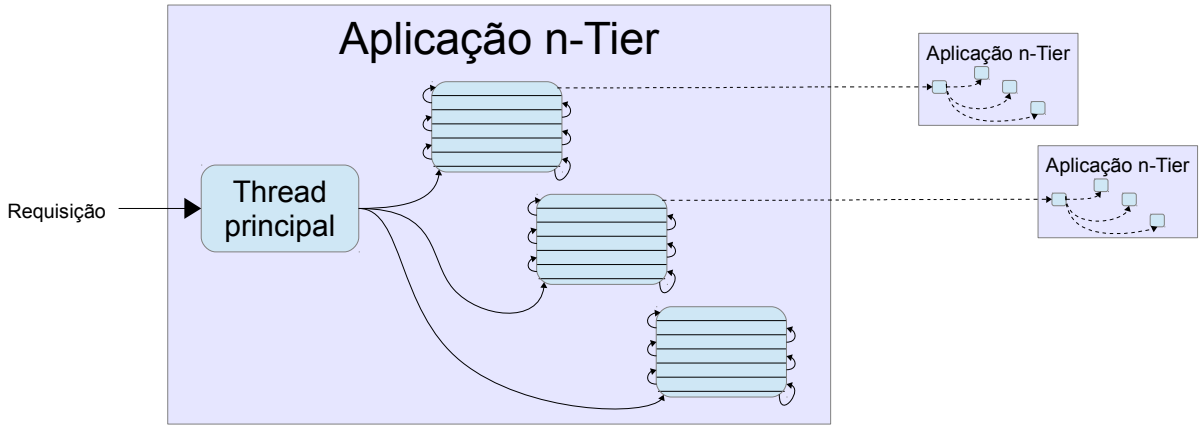


Figura 6.2: Visão geral da aplicação n-Tier. A requisição é recebida por uma *thread* principal e deve ser encaminhada para alguma *thread* do *pool*, ou seja, a primeira camada de processamento.

A aplicação n-Tier utiliza o modelo cliente-servidor em sua implementação, em que cada instância da aplicação produz uma réplica composta por um *pool* de *threads* em que cada *thread* possui diversas camadas de processamento. Dessa forma, cada réplica possui uma *thread* principal que recebe as requisições e as encaminha para uma das *threads* do

pool, ou seja, a primeira camada de processamento.

A primeira camada de processamento, ao receber uma requisição, deve decidir se processa essa requisição ou se realiza um encaminhamento. Caso a decisão seja não processar a requisição, então essa camada deve decidir se encaminha a requisição para uma nova camada ou para uma outra instância da aplicação. Ambas decisões são determinadas aleatoriamente. A cada requisição, a *thread* decide, com base em uma variável aleatória com distribuição uniforme, se a requisição deve ser repassada ou não. A probabilidade de repasse é um parâmetro de execução da aplicação n-Tier.

O mecanismo de IPC `socket` é utilizado para o encaminhamento entre as réplicas, visto que, cada *host* da rede produzida por Mininet possui um endereço IP próprio e executa uma instância da aplicação n-Tier. Os encaminhamentos entre as camadas de processamento podem ser realizados utilizando arquivos, *pipes*, memória compartilhada ou filas de mensagens. O Algoritmo 6.1 ilustra o pseudo-código da implementação da aplicação n-Tier.

Algoritmo 6.1: Função das *threads* da aplicação n-Tier

```
1  início
2      enquanto VERDADEIRO faça
3          req = recebe_requisição();
4          se deve_responder() então
5              responde_requisição(req);
6          senão
7              ipc = escolha_ipc();
8              camada = cria_camada(ipc);
9              encaminha_requisição(camada, req);
10 fim
```

Com a aplicação n-Tier, é possível verificar que uma única requisição pode ser processada por diversas *threads* e processos, além de utilizar vários métodos de IPC. Dessa forma, obter caminhos causais bem definidos de granularidade fina das requisições da aplicação n-Tier é equivalente a obter caminhos causais de um conjunto significativo de aplicações distribuídas.

De acordo com a topologia elaborada para a avaliação e ilustrada na Figura 6.1, os

hosts h_4, h_5, ..., h_{11} executam, individualmente, a aplicação n-Tier. Dessa forma, há oito instâncias da aplicação na rede. Cada *host* possui endereço IP único na mesma sub-rede das demais réplicas. Os agentes da ferramenta S-Trace são instalados em todos os *hosts*. O componente *sniffer* é executado juntamente com a réplica da aplicação e o componente *wrapper* é carregado durante a inicialização da aplicação n-Tier. As requisições são realizadas pelos clientes h_1, h_2 e h_3.

Para mostrar a interação de S-Trace em uma SDN, a técnica de NAT foi implementada nos *switches* of-sw2 e of-sw3 de tal forma que quando réplicas da aplicação n-Tier que estão na mesma sub-rede trocam informações entre si, as conexões são realizadas por meio dos endereços `200.200.200.200` e `200.200.200.201`. Dessa forma, quando os comutadores recebem pacotes de rede com esses endereços de destino, eles são instruídos a modificar o campo destino IP para o endereço da réplica correta. Além disso, o campo origem IP também é modificado para o endereço `200.200.200.200` ou `200.200.200.201` nos pacotes com as respostas das réplicas.

6.1.2 TPC-W

TPC-W [20] é um *benchmark* para transações *Web* do tipo *e-Commerce* que utiliza o servidor *Web* Tomcat (Versão 6.0) [7] e o servidor de banco de dados MySQL (Versão 5.1.63) [6]. Esse *benchmark* realiza diversas requisições ao servidor *Web* para comparar o desempenho do servidor ao processar as requisições. Essas requisições são comuns a vários tipos de sistema de *e-Commerce*, como: navegação pela lista de produtos, compras, vendas, etc. Dessa forma, várias requisições são disparadas, simulando uma utilização real de um sistema *e-Commerce*.

TPC-W foi escolhido para a avaliação por ser uma aplicação efetiva e consolidada. Com isso, é possível verificar como S-Trace se comporta em um ambiente de sistemas reais. Durante a avaliação de S-Trace com a aplicação TPC-W, os *hosts* h_{12} e h_{13} da topologia de avaliação executam as aplicações Tomcat e MySQL, respectivamente.

Para que o componente *wrapper* seja utilizado pela aplicação Tomcat, o arquivo `setenv.sh` do diretório `/usr/share/tomcat6/bin/` deve ser modificado, fazendo com

que a biblioteca *libwrapper* seja carregada na inicialização da aplicação. Para isso a linha `export LD_PRELOAD=‘‘/usr/lib/libwrapper.so’’` foi adicionada a esse arquivo. Dessa forma, quando a aplicação for iniciada, a biblioteca *libwrapper* é carregada antes de qualquer outra biblioteca.

O servidor de banco de dados também precisa ser monitorado pelo agente de S-Trace. MySQL possui uma maneira particular de execução de sua aplicação:1 o usuário `mysql` é criado para instanciar a aplicação. Dessa forma, o *daemon* `mysqld` é executado pelo usuário `mysql`. Portanto, para que a biblioteca *libwrapper* seja carregada na inicialização do *daemon*, a variável de ambiente `LD_PRELOAD` é determinada com o caminho da biblioteca no arquivo `/home/mysql/.bashrc`. Diversos comandos de inicialização para o usuário `mysql` estão agrupados nesse arquivo.

O outro componente do agente, *sniffer*, é executado normalmente pelos *hosts* h_{i2} e h_{i3} para monitorar os pacotes enviados e recebidos pelos servidores.

6.2 Resultados

6.2.1 n-Tier

A aplicação n-Tier foi desenvolvida com intuito de avaliar os caminhos causais construídos por S-Trace. A aplicação foi instrumentada a fim de notificar cada chamada de função realizada pelas *threads* de cada réplica. As notificações de cada réplica são aglutinadas em um único arquivo, denominado arquivo de traço, com a seguinte estrutura:

```
(Carimbo de Tempo) (IP: endereço_réplica) [LWPID/PID] - função 1
(Carimbo de Tempo) (IP: endereço_réplica) [LWPID/PID] - função 2
(Carimbo de Tempo) (IP: endereço_réplica) [LWPID/PID] - função 3
...
(Carimbo de Tempo) (IP: endereço_réplica) [LWPID/PID] - função n
```

O arquivo de traço possui todas as chamadas de função utilizadas pelas réplicas para processar as requisições dos clientes. Cada entrada desse arquivo possui as seguintes informações: carimbo de tempo indicando o momento em que a função foi invocada, endereço IP da réplica da aplicação n-Tier, LWPID e PID de quem invocou a função

e, por fim, a função invocada. Esse arquivo é constrastado com os caminhos causais construídos por S-Trace para verificar a corretude e precisão dos caminhos.

Os caminhos causais construídos por S-Trace são ilustrados em forma de grafo em que cada nó representa uma função que foi invocada por uma *thread* de uma aplicação e também um pacote que trafegou na rede proveniente da chamada da função. As arestas do grafo indicam a relação de ordem entre os nós, ou seja, uma aresta que parte de um nó A para um nó B indica que B ocorreu depois de A. A Figura 6.3 apresenta a estrutura do grafo produzido por S-Trace.

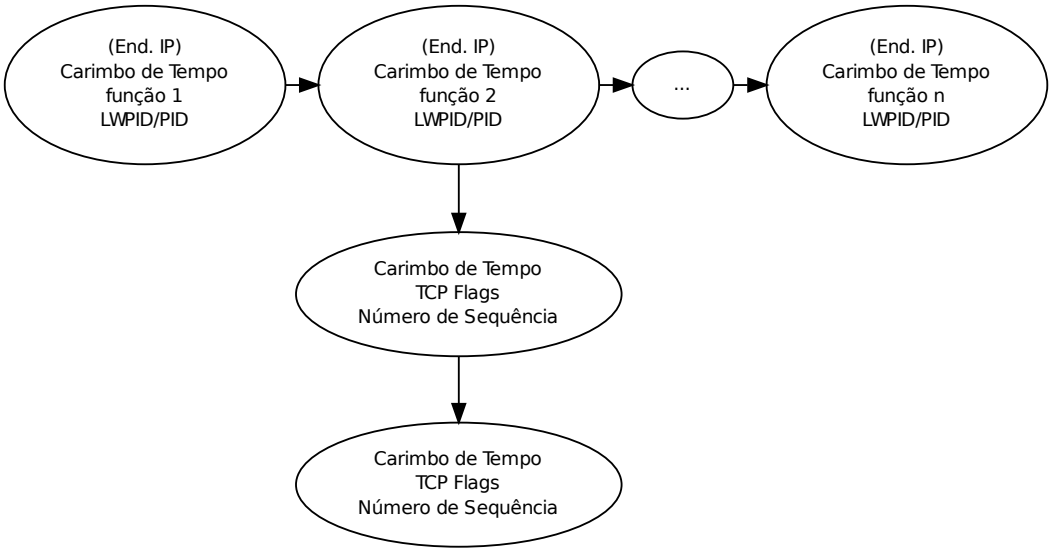


Figura 6.3: Estrutura utilizada na visualização dos caminhos causais. Os nós representam funções ou pacotes e as arestas representam a relação de ordem entre os nós.

Os nós em nível horizontal representam as chamadas de função e possuem as seguintes informações: endereço IP do agente que enviou os dados para o gerente, carimbo de tempo, identificação da função invocada, LWPID e PID. Os nós em nível vertical representam os pacotes de rede que trafegaram devido à função invocada. Estes nós possuem as seguintes informações: carimbo de tempo, flags do protocolo TCP, número de sequência do pacote e indicação de erro, caso exista.

Para avaliar a corretude e a precisão dos caminhos causais construídos por S-Trace, uma requisição com destino ao endereço IP `200.200.200.200` é realizada pelo *host* h_2. Essa requisição pode ser processada por uma única réplica ou por um conjunto de réplicas. As chamadas de função presente no arquivo de traço são mostradas a seguir:

```
(1404430317.081565) (10.0.0.4) [2021/1970] - accept
(1404430317.081883) (10.0.0.4) [2021/1970] - recv
(1404430317.933361) (10.0.0.4) [2021/1970] - socket
(1404430317.934326) (10.0.0.4) [2021/1970] - connect(10.0.0.5:3333)
(1404430317.944676) (10.0.0.4) [2021/1970] - send
(1404430317.945154) (10.0.0.5) [2052/1984] - accept
(1404430317.945834) (10.0.0.5) [2052/1984] - recv
(1404430317.947011) (10.0.0.4) [2021/1970] - recv
(1404430317.947388) (10.0.0.5) [2052/1984] - shmget
(1404430317.949425) (10.0.0.5) [2052/1984] - shmat
(1404430317.950733) (10.0.0.5) [2176/1984] - shmget
(1404430317.952657) (10.0.0.5) [2176/1984] - shmat
(1404430317.954214) (10.0.0.5) [2176/1984] - shmget
(1404430317.956379) (10.0.0.5) [2176/1984] - shmat
(1404430317.958091) (10.0.0.5) [2180/1984] - shmget
(1404430317.959526) (10.0.0.5) [2180/1984] - shmat
(1404430317.961263) (10.0.0.5) [2176/1984] - socket
(1404430317.962918) (10.0.0.5) [2176/1984] - connect(200.200.200.201:3335)
(1404430317.977137) (10.0.0.5) [2176/1984] - send
(1404430317.977244) (10.0.0.7) [2061/2027] - accept
(1404430317.977744) (10.0.0.7) [2061/2027] - recv
(1404430317.979610) (10.0.0.5) [2176/1984] - recv
(1404430317.980650) (10.0.0.7) [2061/2027] - shmget
(1404430317.982121) (10.0.0.7) [2061/2027] - shmat
(1404430317.983756) (10.0.0.7) [2191/2027] - shmget
(1404430317.985230) (10.0.0.7) [2191/2027] - shmat
(1404430317.986744) (10.0.0.7) [2191/2027] - socket
(1404430317.989014) (10.0.0.7) [2191/2027] - connect(10.0.0.6:3334)
(1404430318.002498) (10.0.0.6) [2047/2006] - accept
(1404430318.003499) (10.0.0.7) [2191/2027] - send
(1404430318.004067) (10.0.0.6) [2047/2006] - recv
(1404430318.006001) (10.0.0.6) [2047/2006] - send
(1404430318.006893) (10.0.0.7) [2191/2027] - recv
(1404430318.008360) (10.0.0.6) [2047/2006] - close
(1404430318.008838) (10.0.0.7) [2191/2027] - close
(1404430318.010795) (10.0.0.7) [2061/2027] - send
(1404430318.017404) (10.0.0.7) [2061/2027] - close
(1404430318.017637) (10.0.0.5) [2176/1984] - close
(1404430318.020688) (10.0.0.5) [2052/1984] - send
(1404430318.022683) (10.0.0.4) [2021/1970] - close
(1404430318.022820) (10.0.0.5) [2052/1984] - close
(1404430318.024599) (10.0.0.4) [2021/1970] - send
(1404430318.026886) (10.0.0.4) [2021/1970] - close
```

Após processar toda a requisição, o processo de construção dos caminhos causais é acionado e os caminhos causais iniciais da requisição são criados. Alguns nós foram ocultados para facilitar a visualização dos caminhos. A Figura 6.4 ilustra o grafo construído.

S-Trace produziu dois caminhos causais, indicando que foram processadas duas requisições entre as réplicas, o que está incorreto. Esse fato acontece porque não há

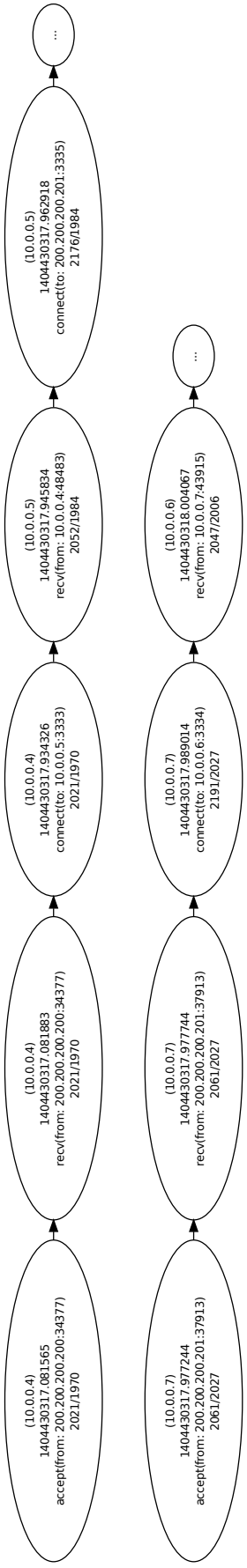


Figura 6.4: Caminhos causais de uma única requisição da aplicação n-Tier. A ferramenta, no processo inicial, produziu dois caminhos causais, indicando que foram duas requisições, o que está incorreto. Esses caminhos foram construídos antes do processo de reprodução do tráfego de rede que aprimora esses caminhos. Alguns nós do grafo foram ocultados para facilitar a visualização.

nenhum tipo de ligação entre os nós dos caminhos causais. Para contornar essa situação, o gerente de reprodução recebe os nós de início de conexão e executa a técnica de reprodução do tráfego de rede para observar o comportamento da rede. A Figura 6.5 ilustra o grafo produzido pela ferramenta após a reprodução do tráfego. A partir do arquivo de traço e o caminho causal produzido pela ferramenta, é possível observar que o comportamento da aplicação foi totalmente capturado e ilustrado pelo caminho causal produzido. Os nós referentes aos pacotes de rede foram ocultados para facilitar a visualização.

O caminho causal construído pela ferramenta além de auxiliar a compreender o comportamento das aplicações, facilita a detecção de anomalias. Duas falhas foram injetadas nas réplicas da aplicação n-Tier para demonstrar que os caminhos causais auxiliam na detecção de anomalias. A primeira falha faz uma *thread* adormecer por um período grande, simulando uma aplicação sobrecarregada. A outra falha faz uma *thread* realizar uma conexão com uma réplica inexistente, ou seja, com um endereço IP desconhecido.

A primeira falha injetada foi introduzida para simular um comportamento não esperado pela aplicação, fazendo com que uma *thread* adormeça por dois minutos. Essa falha remete à ideia de que o processamento gastou um tempo maior que o esperado ou que realmente aconteceu uma falha leve no processamento. Dessa forma, o processamento dessa requisição demanda muito tempo para ser realizado. A partir do caminho causal construído por S-Trace, é possível determinar onde ocorreu esse atraso e qual *thread* foi responsável. A Figura 6.6 ilustra o caminho causal dessa requisição.

A partir do caminho produzido pela ferramenta, é possível identificar que a requisição foi processada por uma única réplica (`10.0.0.9`). É possível notar, também, que o processamento gastou um tempo maior que o esperado. Esse fato é identificado pela diferença entre os carimbos de tempo dos nós. Para mostrar isso, a aresta que liga esses nós é ilustrada de forma pontilhada. A ferramenta, ao adicionar uma aresta entre dois nós, verifica se a diferença de tempo é maior do que um minuto (ou um parâmetro predefinido), e em caso positivo, a aresta é desenhada com uma linha pontilhada.

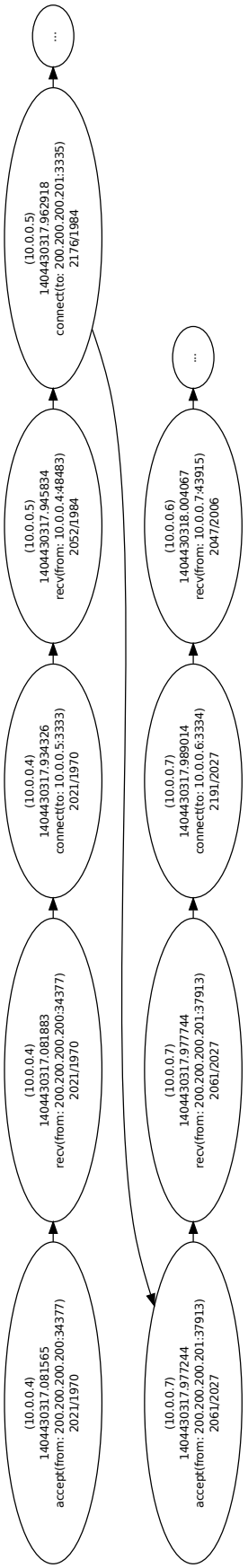


Figura 6.5: Caminho causal da mesma requisição da aplicação n-Tier ilustrada na Figura 6.4. A ferramenta inicialmente produziu dois caminhos causais, mas após a reprodução de tráfego, foi descoberta uma ligação entre os nós de início de conexão e, com isso, os caminhos foram aglutinados, indicando que todas as operações foram realizadas para processar a única requisição.

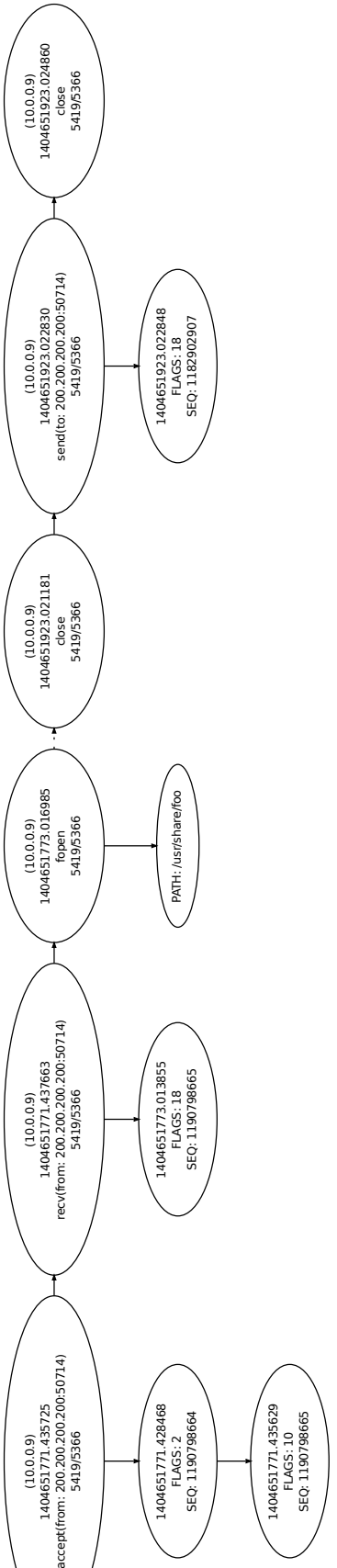


Figura 6.6: Caminho causal de uma requisição da aplicação n-Tier com a falha de adormecer injetada. A partir desse caminho causal, é possível inferir que o processamento foi realizado por uma única *thread* e que após obter o descritor do arquivo `/usr/share/foo` até sua próxima chamada de função, foram gastos, aproximadamente, 150 segundos, dessa forma, a aresta de ligação é representada com uma linha pontilhada, pois o tempo entre os nós foi maior que 60 segundos.

A outra falha injetada faz com que uma réplica tente realizar uma conexão com uma instância inexistente, ou seja, uma réplica com endereço desconhecido. Para isso, uma determinada réplica foi modificada para realizar conexão com uma réplica com endereço IP `200.200.200.210` ao invés de `200.200.200.201`. Dessa forma, a conexão não é concretizada e a execução da aplicação fica comprometida. A Figura 6.7 ilustra o caminho causal dessa requisição construído por S-Trace.

A partir do caminho causal construído, é possível verificar que a função `connect` gerou um erro e nenhum pacote foi trafegado. Esse erro foi provocado por uma tentativa de conexão a um endereço desconhecido. Com isso, a execução da aplicação foi encerrada de maneira não esperada.

A Tabela 6.1 ilustra a quantidade de cada mecanismo de IPC utilizado em 200 execuções da aplicação n-Tier. Os mecanismos são: arquivo, *pipe*, memória compartilhada, fila de mensagens e *socket*. Claramente, o valor da somatória das quantidades de cada mecanismo de IPC é superior ao valor 200, isso se deve ao fato de que para processar uma única requisição, diversos mecanismos podem ser utilizados.

Tabela 6.1: Quantidade de cada mecanismo de IPC utilizado pela aplicação n-Tier para processar 200 requisições.

Arquivo	*Pipe*	Memória compartilhada	Fila de mensagens	*Socket*
61	63	43	52	45

6.2.2 TPC-W

A ferramenta S-Trace foi também avaliada com a aplicação TPC-W. Nessa avaliação, os *hosts* h_1, h_2 e h_3 fazem requisições diretamente para o *host* h_{12} (servidor *Web*) que, por sua vez, realiza requisições para o *host* h_{13} (servidor de banco de dados) para completar o processamento da requisição inicial. Como TPC-W é uma aplicação real, ela não foi instrumentada e, com isso, o arquivo de traço não é disponibilizado.

A Figura 6.8 ilustra um caminho causal construído por S-Trace para requisições da aplicação TPC-W. Esse caminho foi modificado, fazendo com que alguns nós sejam ocultados para facilitar a visualização. A partir desse caminho, é possível verificar que o

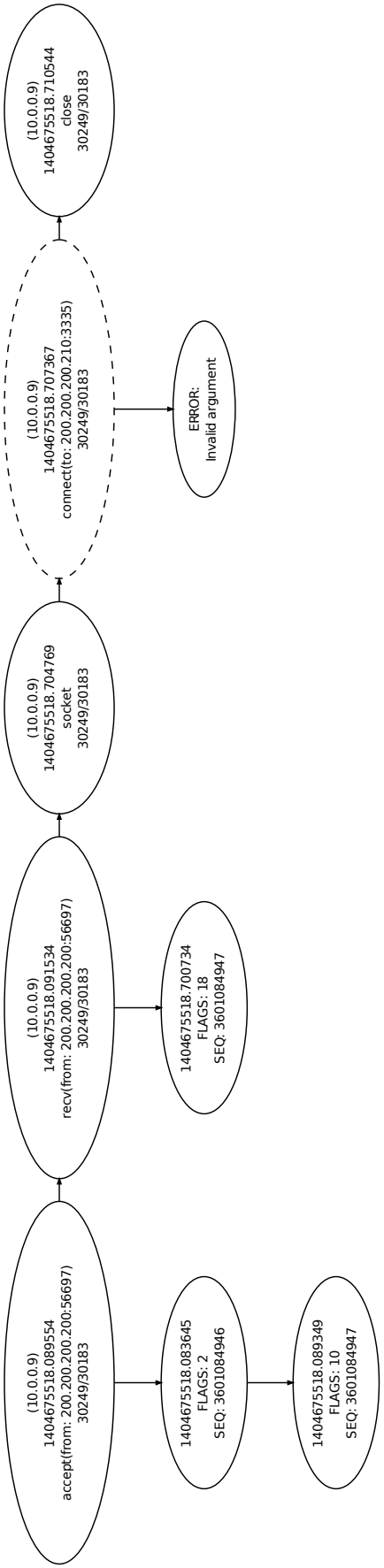


Figura 6.7: Caminho causal de uma requisição da aplicação n-Tier com a falha de erro de conexão. O caminho indica que a *thread* 30249 recebeu a requisição e tentou conexão com o endereço 200.200.200.210 e obteve um erro. Os nós que notificam erros são tracejados para auxiliar a visualização.

Figura 6.8: Caminho causal de uma requisição da aplicação TPC-W que necessitou conexão com o banco de dados. O caminho indica que a *thread* 10737 recebe a requisição do cliente, realiza a abertura de diversos arquivos, entre eles, um arquivo de configuração para a conexão com o banco de dados, conecta com o servidor MySQL e, por fim, envia uma nova requisição para este servidor. Esta última operação foi realizada pela função `send` e gerou diversos pacotes de rede.

processamento da requisição necessitou abertura de diversos arquivos e conexão com o servidor de banco de dados. Nessa conexão, uma nova requisição é realizada para o MySQL, gerando diversos pacotes de rede. Dessa forma, é possível inferir que a requisição foi processada normalmente, sem nenhum indicativo de erro.

Durante a avaliação, o servidor de banco de dados foi desativado para ilustrar como seria o comportamento da aplicação sem conexão com o servidor MySQL. Na execução, algumas requisições não obtiveram respostas pois necessitavam de conexões com o banco, enquanto outras obtiveram respostas pois o processamento era local. A Figura 6.9 ilustra dois caminhos causais sendo que o caminho superior ilustra uma requisição que necessitou de conexão com o servidor MySQL, que não foi realizada e o caminho inferior ilustra uma requisição que obteve resposta do servidor Tomcat. Contudo, a última requisição foi processada de forma direta, indicando que o conteúdo da resposta do cliente já tinha sido armazenado na memória. Com isso, não precisou de chamadas de função para obter os dados requisitados.

Os caminhos causais construídos por S-Tracea partir das requisições das aplicações n-Tier e TPC-W mostram que a ferramenta captura o comportamento dessas aplicações de maneira confiável, ou seja, as interações realizadas no processamento das requisições são capturadas e aglutinadas em caminhos causais bem definidos.

6.2.3 *wrapper*

O componente *wrapper* do agente da ferramenta S-Trace também foi avaliado para demonstrar o impacto, em tempo de execução, da utilização do agente nas aplicações, mais especificamente, a utilização da biblioteca *libwrapper*. Para isso, uma aplicação cliente/servidor simples foi desenvolvida em que o cliente realiza uma conexão com o servidor, envia um *buffer* de mensagem e encerra a conexão. Além disso, durante a execução dessa aplicação, as funções `fork` e `fopen` foram invocadas. O Algoritmo 6.2 ilustra um trecho da implementação do cliente.

Na avaliação de tempo de execução, é capturado o carimbo de tempo antes e depois da chamada das funções `send`, `fopen` e `fork`. O tamanho do *buffer* da mensagem a ser

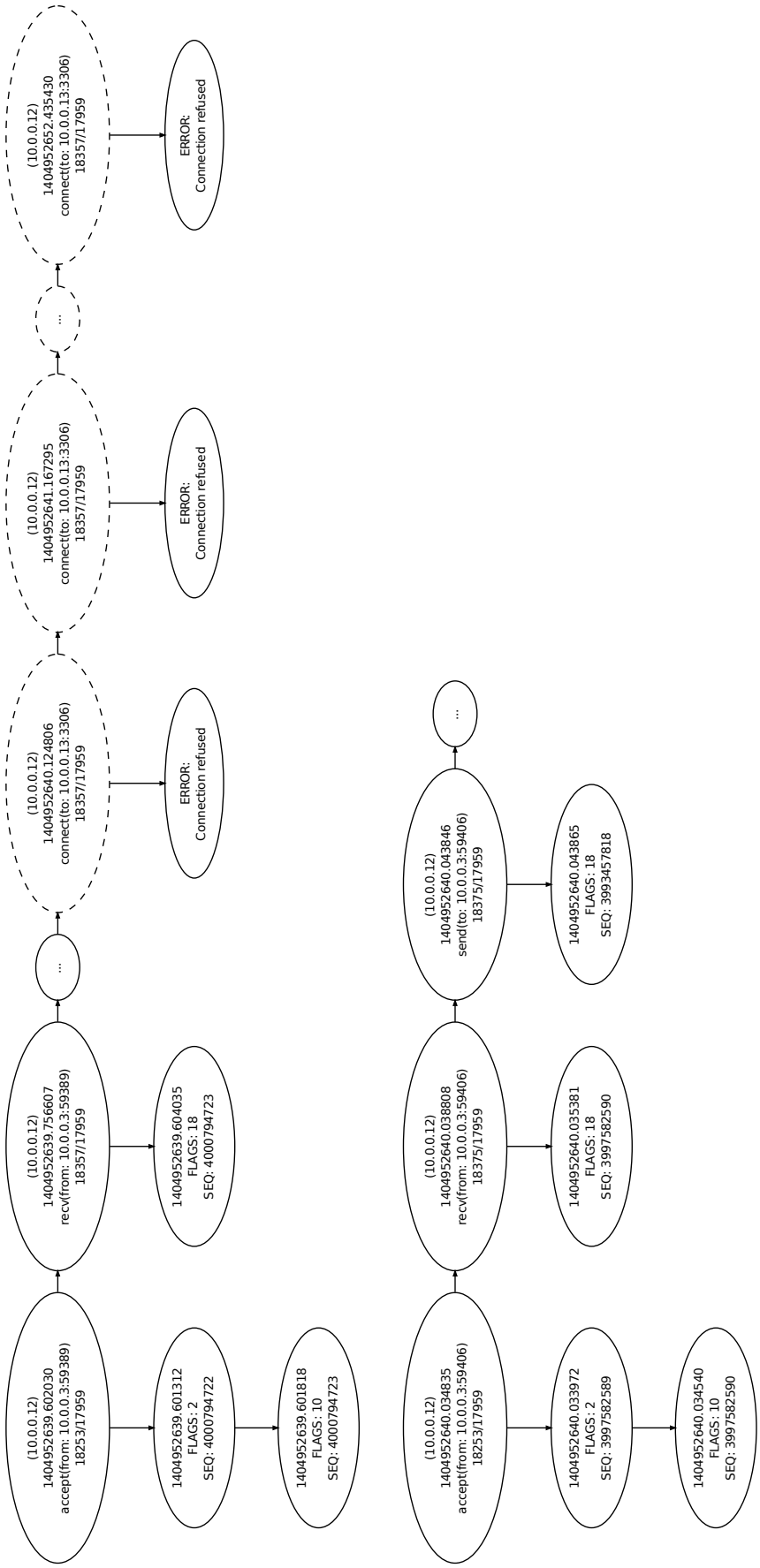


Figura 6.9: Caminhos causais de requisições da aplicação TPC-W em que o servidor de banco de dados foi desativado durante a execução. O caminho superior indica que a *thread*18357 recebeu uma requisição e, por várias vezes, tentou conexão com o servidor de banco de dados sem resposta. O caminho inferior indica que o mesmo cliente realizou uma nova requisição e que foi processada pela mesma *thread* sem nenhum problema.

Algoritmo 6.2: Implementação do cliente

```
1 início
2 |  ...
3 |  connect(sockfd, (struct sockaddr*) &addr, len);
4 |  gettimeofday(&tv1, NULL);
5 |  send(sockfd, buffer, buflen, 0);
6 |  gettimeofday(&tv2, NULL);
7 |  close(sockfd);
8 |  ...
9 fim
```

enviada variou entre 100, 1500 e 10000 bytes. Para cada tamanho de *buffer* utilizado na função `send` e para as funções `fork` e `fopen`, foram executadas 500 instâncias do cliente, produzindo 500 requisições. Essa avaliação foi executada na mesma topologia utilizada nas avaliações anteriores e os *hosts* h_4 e h_7 executaram o cliente e o servidor, respectivamente. A Tabela 6.2 ilustra os resultados obtidos e os intervalos de confiança de 95% do tempo médio.

Tabela 6.2: Tempo de execução médio de chamadas de função (ms).

	sem *wrapper* (I.C. 95%)	com *wrapper* (I.C. 95%)	*overhead*
send (100 bytes)	0,068±0,005	0,144±0,012	~0,076
send (1500 bytes)	0,103±0,009	0,174±0,015	~0,071
send (10000 bytes)	0,150±0,013	0,221±0,019	~0,071
fopen	0,326±0,028	0,401±0,035	~0,075
fork	0,477±0,041	0,553±0,048	~0,076

A partir dos resultados obtidos, é possível verificar que o impacto do componente *wrapper* é desprezível.

6.3 Considerações Finais

Este capítulo apresenta as avaliações realizadas da ferramenta S-Trace e os resultados obtidos. As aplicações utilizadas foram escolhidas para demonstrar a capacidade da ferramenta para construir caminhos causais corretos com diferentes tipos de aplicação e, consequentemente, diferentes modelos de programação. A aplicação n-Tier utiliza diferentes tipos de mecanismos de IPC e foi instrumentada para fornecer um objeto

resposta para comparar com os caminhos causais construídos. A aplicação TPC-W é uma aplicação que simula um comportamento real de um sistema *e-Commerce*, e utiliza servidores comuns no ambiente de aplicações distribuídas, no caso, os servidores Tomcat e MySQL. Além disso, foi avaliado o *overhead* da utilização dos agentes instalados nas estações de trabalhos e servidores. Esses agentes auxiliam os gerentes a construir os caminhos causais e geram *overheads* desprezíveis.

Capítulo 7

Conclusão

O processo de construção de caminhos causais é um fator importante para as ferramentas de detecção de anomalias, uma vez que, a partir de caminhos causais bem definidos, a ferramenta infere o elemento do caminho que está causando a anomalia. Dessa forma, diferentes técnicas para construir os caminhos causais são explorados pela comunidade acadêmica e indústria, bem como, novas técnicas de inferência.

A ferramenta S-Trace foi desenvolvida com intuito de auxiliar a compreensão do comportamento das aplicações e das ferramentas de detecção de anomalias com caminhos causais bem definidos. Esses caminhos são construídos sem que aplicações, *frameworks* ou sistemas operacionais sejam modificados, como as ferramentas não intrusivas. Porém, os caminhos são construídos com alta precisão, como ocorre com as ferramentas instrusivas. Além disso, S-Trace explora as arquiteturas programáveis de separação de planos, mais especificamente, o padrão OpenFlow [21], para aprimorar os caminhos causais construídos pela ferramenta com as técnicas de gravação e reprodução do tráfego de rede.

Os resultados da avaliação mostram que a ferramenta construiu os caminhos causais corretamente, se comparado com aplicação que foi desenvolvida e instrumentada especificamente para a avaliação. Essa aplicação busca simular o comportamento de grande parte das aplicações distribuídas. Além disso, a ferramenta foi avaliada utilizando uma aplicação real que simula o comportamento de um sistema *e-Commerce*

com os servidores *Web* e de banco de dados usados em diversas aplicações de rede.

7.1 Trabalhos Futuros

S-Trace é uma ferramenta em desenvolvimento e ainda sem um módulo de visualização mais robusto. S-Trace gera um arquivo `.dot` que representa os caminhos causais em forma de grafos estáticos. Um caminho com quantidade elevada de elementos é representado em um único grafo de tamanho exorbitante. O módulo de visualização é o próximo passo do trabalho atual. Ele fornecerá uma interface gráfica de tempo real em que administradores de rede possam visualizar centenas de caminhos de forma reduzida e expandi-los para uma análise individualizada do processamento da requisição.

Diversos aspectos do paradigma de Redes Definidas por Software foram inexplorados por S-Trace, tais como: *pipeline* com múltiplas tabelas de fluxos, abstração de grupo, campos flexíveis, etc. Essas novas funcionalidades abrem novos caminhos que podem ser explorados por ferramentas como S-Trace, por exemplo, incorporar as múltiplas tabelas de fluxos aos caminhos causais construídos.

Referências Bibliográficas

[1] GCC. http://gcc.gnu.org. Acessado em 01/07/2014.

[2] Nagios. http://www.nagios.com. Acessado em 01/07/2014.

[3] Icinga. http://www.icinga.com. Acessado em 01/07/2014.

[4] Zenoss. http://www.zenoss.com. Acessado em 01/07/2014.

[5] Octave. http://www.octave.org. Acessado em 01/07/2014.

[6] MySQL. http://www.mysql.com. Acessado em 01/07/2014.

[7] Tomcat. http://tomcat.apache.org. Acessado em 01/07/2014.

[8] Graphviz. http://www.graphviz.org. Acessado em 15/06/2013.

[9] Jetty. http://www.eclipse.org/jetty. Acessado em 15/06/2013.

[10] CoralCDN. http://www.coralcdn.org. Acessado em 28/05/2013.

[11] Mininet. http://mininet.org/download. Acessado em 01/07/2014.

[12] *libc*. http://www.gnu.org/software/libc. Acessado em 05/06/2013.

[13] VirtualBox. https://www.virtualbox.org. Acessado em 01/07/2014.

[14] Justin Scheck. Taming Technology Sprawl. TheWallStreetJournal. Acessado em 18/07/2013.

[15] OpenFlow Specification 1.0.0. http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf. Acessado em 01/07/2014.

[16] OpenFlow Specification *1.1.0*. `http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf`. Acessado em 11/06/2013.

[17] Software-Defined Networking: The New Norm of Networks. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf`. Acessado em 01/07/2014.

[18] OFRewind Legacy Research Project Files. `http://www1.icsi.berkeley.edu/~andi/ofrewind`. Acessado em 01/07/2014.

[19] Open Networking Foundation. SDN Architecture. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR\_SDN\_ARCH\_1.0\_06062014.pdf`. Acessado em 01/07/2014.

[20] Daniel Menasce. TPC-W: a Benchmark for *e-Commerce*. *Internet Computing, IEEE*, páginas 83–87, 2002.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker e Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, páginas 69–74, 2008.

[22] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown e Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, páginas 105–110, 2008.

[23] Paul Barham, Richard Black, Moises Goldszmidt, Rebecca Isaacs, John MacCormick, Richard Mortier e Aleksandr Simma. Constellation: Automated Discovery of Service and Host Dependencies in Networked Systems. Relatório Técnico MSR-TR-2008-67, Microsoft Research, 2008.

[24] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown e Guru Parulkar. FlowVisor: A Network Virtualization Layer. Relatório Técnico TR09-01, Stanford University, 2009.

[25] Zheng Cai, Alan L. Cox e T. S. Eugene Ng. Maestro: A System for Scalable OpenFlow Control. Relatório Técnico TR10-08, Rice University, 2010.

[26] William Richard Stevens, Bill Fenner e Andrew M. Rudoff. *UNIX Network Programming*. Pearson Education, 3 edição, 2003. ISBN 0131411551.

[27] William Richard Stevens. *Advanced Programming in the UNIX® Environment*. Pearson Education, LPE edição, 2004. ISBN 8178080966.

[28] Abraham Silberschatz, Peter Baer Galvin e Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8 edição, 2008. ISBN 0470128720.

[29] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds e Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. Em *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOPS'03)*. 2003.

[30] Paul Barham, Austin Donnelly, Rebecca Isaacs e Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. Em *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*. 2004.

[31] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox e Eric Brewer. Path-Based Failure and Evolution Management. Em *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. 2004.

[32] Michael J. Freedman, Karthik Lakshminarayanan e David Mazières. OASIS: Anycast for Any Service. Em *Proceedings of the 3rd Symposium on Networked Systems Design & Implementation (NSDI'06)*. 2006.

[33] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera e Amin Vahdat. Wap5: Black-box performance debugging for wide-area systems. Em *Proceedings of the 15th International Conference on World Wide Web (WWW'06)*. 2006.

[34] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz e Ming 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)*. 2007.

[35] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker e Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. Em *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI'07)*. 2007.

[36] Srikanth Kandula, Ranveer Chandra e Dina Katabi. What's going on?: Learning communication rules in edge networks. Em *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM'08)*. 2008.

[37] Eric Koskinen e John Jannotti. BorderPatrol: Isolating Events for Black-box Tracing. Em *Proceedings of the 3rd EuroSys European Conference on Computer Systems (Eurosys'08)*. 2008.

[38] Lucian Popa, Byung-Gon Chun, Ion Stoica, Jaideep Chandrashekar e Nina Taft. Macroscope: End-Point Approach to Networked Application Dependency Discovery. Em *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'09)*. 2009.

[39] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye e Paramvir 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)*. 2009.

[40] Bob Lantz, Brandon Heller e Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. Em *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IX)*. 2010.

[41] Rodrigo Fonseca, Michael J. Freedman e George Porter. Experiences with Tracing Causality in Networked Services. Em *Proceedings of the Internet Network*

Management Conference on Research on Enterprise Networking (INM/WREN'10). 2010.

[42] Dipu John, Pawan Prakash, Ramana Rao Kompella e Ranveer Chandra. Shedding Light on Enterprise Network Failures Using Spotlight. Em *Proceedings of the 29th Symposium on Reliable Distributed Systems (SRDS'10)*. 2010.

[43] Andreas Wundsam, Dan Levin, Srini Seetharaman e Anja Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. Em *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. 2011.

[44] Arun Natarajan, Peng Ning, Yao Liu, Sushil Jajodia e Steve E. Hutchinson. NSDMiner: Automated discovery of Network Service Dependencies. Em *Proceedings of the 31st Annual Internation Conference on Computer Communications (INFOCOM'12)*. 2012.

[45] David Erickson. The Beacon Openflow Controller. Em *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*. 2013.

[46] Brivaldo. A. S. Júnior, Fabrício B. de Carvalho e Ronaldo A. Ferreira. Nemo: Procurando e Encontrando Anomalias em Aplicações Distribuídas. Em *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'13)*. 2013.

Apêndice A

Funções *wrappers*

Este apêndice apresenta todas as funções interceptadas pelo componente *wrapper* da ferramenta S-Trace. Para facilitar a localização, as funções estão listadas em ordem alfabética.

```
//accept
int accept(int sockfd, struct sockaddr* addr, socklen_t* addrlen) {
  int retvalue;
  struct timeval tv;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*accept_real)(int, struct sockaddr*, socklen_t*) = NULL;

  if(!accept_real)
    accept_real = dlsym(RTLD_NEXT, "accept");

  retvalue = accept_real(sockfd, addr, addrlen);

  gettimeofday(&tv, NULL);
  fstat(sockfd, &statbuf);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(sockfd);
  data->arg2 = htonl(retvalue);
  data->op   = htons(145);

  if(S_ISSOCK(statbuf.st_mode)) {
    socklen_t len;
    struct sockaddr_storage addr;

    len = sizeof(addr);
    if(getpeername(retvalue, (struct sockaddr*) &addr, &len) == 0) {
      if(addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
```

```
        data->source_port     = s->sin_port;
        data->source_ip       = s->sin_addr;
        data->arg3 = htonl(0);
      }
    }
    if(getsockname(retvalue, (struct sockaddr*) &addr, &len) == 0) {
      if(addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->dest_port       = s->sin_port;
        data->dest_ip         = s->sin_addr;
        data->arg3 = htonl(0);
      } else if(addr.ss_family == AF_FILE || addr.ss_family == PF_FILE) {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }
  } else if(S_ISFIFO(statbuf.st_mode)) {
    data->arg3 = htonl(2);
  } else {
    free(data);
    return retvalue;
  }

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//bind
int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*bind_real)(int, const struct sockaddr*, socklen_t) = NULL;

  if(!bind_real)
    bind_real = dlsym(RTLD_NEXT, "bind");

  retvalue = bind_real(sockfd, addr, addrlen);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(sockfd);

  if(addr->sa_family == AF_INET) {
    socklen_t len;
    struct sockaddr_storage addr;
    struct sockaddr_in* s = (struct sockaddr_in*) &addr;

    data->source_port = s->sin_port;
    data->source_ip   = s->sin_addr;
    data->op = htons(142);
  } else if(addr->sa_family == AF_UNIX) {
    strcpy(data->path, ((struct sockaddr_un*) &addr)->sun_path);
    data->op = htons(143);
  } else {
```

```
    free(data);
    return retvalue;
  }

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//close
int close(int fd) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*close_real)(int) = NULL;

  if(!close_real)
    close_real = dlsym(RTLD_NEXT, "close");

  retvalue = close_real(fd);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(fd);
  data->arg2 = htonl(retvalue);
  data->op   = htons(147);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//connect
int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*connect_real)(int, const struct sockaddr*, socklen_t) = NULL;

  if(!connect_real)
    connect_real = dlsym(RTLD_NEXT, "connect");

  retvalue = connect_real(sockfd, addr, addrlen);

  fstat(sockfd, &statbuf);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
```

```
  data->arg1 = htonl(sockfd);
  data->arg2 = htonl(retvalue);
  data->op   = htons(146);

  if(S_ISSOCK(statbuf.st_mode)) {
    if(retvalue < 0) {
      struct sockaddr_in* s = (struct sockaddr_in*) addr;
      data->dest_port       = s->sin_port;
      data->dest_ip         = s->sin_addr;
      strcpy(data->path, strerror(errno));
    } else {
      socklen_t len;
      struct sockaddr_storage addr;

      len = sizeof(addr);
      if(getpeername(sockfd, (struct sockaddr*) &addr, &len) == 0) {
        if (addr.ss_family == AF_INET) {
          struct sockaddr_in* s = (struct sockaddr_in*) &addr;
          data->dest_port     = s->sin_port;
          data->dest_ip       = s->sin_addr;
          data->arg3 = htonl(0);
        } else if(addr.ss_family == AF_FILE) {
          struct sockaddr_un* s = (struct sockaddr_un*) &addr;
          strcpy(data->path, s->sun_path);
          data->arg3 = htonl(1);
        }
      }
      if(getsockname(sockfd, (struct sockaddr*) &addr, &len) == 0) {
        if (addr.ss_family == AF_INET) {
          struct sockaddr_in* s = (struct sockaddr_in*) &addr;
          data->source_port      = s->sin_port;
          data->source_ip        = s->sin_addr;
          data->arg3 = htonl(0);
        }
      }
    }
  }

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//dup
int dup(int oldfd) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*dup_real)(int) = NULL;

  if(!dup_real)
    dup_real = dlsym(RTLD_NEXT, "dup");

  retvalue = dup_real(oldfd);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
```

```
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(oldfd);
  data->arg2 = htonl(retvalue);
  data->op   = htons(161);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//dup2
int dup2(int oldfd, int newfd) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*dup2_real)(int, int) = NULL;

  if(!dup2_real)
    dup2_real = dlsym(RTLD_NEXT, "dup2");

  retvalue = dup2_real(oldfd, newfd);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(oldfd);
  data->arg2 = htonl(newfd);
  data->arg3 = htonl(retvalue);
  data->op   = htons(162);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//fclose
int fclose(FILE* fp) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int fd;
  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*fclose_real)(FILE*) = NULL;

  if(!fclose_real)
    fclose_real = dlsym(RTLD_NEXT, "fclose");

  if(fp != NULL)
    fd = fp->_fileno;

  retvalue = fclose_real(fp);

  if(fp != NULL) {
    data->source_port = data->dest_port = 0;
    data->lwp  = htons(syscall(SYS_gettid));
```

```
    data->pid  = htons(syscall(SYS_getpid));
    data->ppid = htons(syscall(SYS_getppid));
    data->mlen = htonl(0);
    data->sec  = htonl(tv.tv_sec);
    data->usec = htonl(tv.tv_usec);
    data->arg1 = htonl(fd);
    data->op   = htons(152);

    send_manager(data, (void*) NULL, 0);
  }

  free(data);

  return retvalue;
}


//fopen
FILE* fopen(const char* pathname, const char* mode) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  FILE* retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static FILE* (*fopen_real)(const char*, const char*) = NULL;

  if(!fopen_real)
    fopen_real = dlsym(RTLD_NEXT, "fopen");

  retvalue = fopen_real(pathname, mode);

  if(retvalue != NULL) {
    fstat(retvalue->_fileno, &statbuf);

    data->source_port = data->dest_port = 0;
    data->lwp  = htons(syscall(SYS_gettid));
    data->pid  = htons(syscall(SYS_getpid));
    data->ppid = htons(syscall(SYS_getppid));
    data->mlen = htonl(0);
    data->sec  = htonl(tv.tv_sec);
    data->usec = htonl(tv.tv_usec);
    data->arg1 = htonl(retvalue->_fileno);
    data->op   = htons(151);
    strcpy(data->path, pathname);

    send_manager(data, (void*) NULL, 0);
  }

  free(data);

  return retvalue;
}


//fork
pid_t fork(void) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  pid_t retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static pid_t (*fork_real)(void) = NULL;

  if(!fork_real)
    fork_real = dlsym(RTLD_NEXT, "fork");
```

```
  retvalue = fork_real();

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(retvalue);
  data->op   = htons(171);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//listen
int listen(int sockfd, int backlog) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*listen_real)(int, int) = NULL;

  if(!listen_real)
    listen_real = dlsym(RTLD_NEXT, "listen");

  retvalue = listen_real(sockfd, backlog);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(sockfd);
  data->op   = htons(144);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//mkfifo
int mkfifo(const char* pathname, mode_t mode) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*mkfifo_real)(const char*, mode_t) = NULL;

  if(!mkfifo_real)
    mkfifo_real = dlsym(RTLD_NEXT, "mkfifo");

  retvalue = mkfifo_real(pathname, mode);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
```

```
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(mode);
  data->op = htons(221);
  strcpy(data->path, pathname);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//msgget
int msgget(key_t key, int msgflg) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*msgget_real)(key_t, int) = NULL;

  if(!msgget_real)
    msgget_real = dlsym(RTLD_NEXT, "msgget");

  retvalue = msgget_real(key, msgflg);

  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(key);
  data->arg2 = htonl(msgflg);
  data->arg3 = htonl(retvalue);
  data->op   = htons(181);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//msgrcv
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  ssize_t retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static ssize_t (*msgrcv_real)(int, void*, size_t, long, int) = NULL;

  if(!msgrcv_real)
    msgrcv_real = dlsym(RTLD_NEXT, "msgrcv");

  retvalue = msgrcv_real(msqid, msgp, msgsz, msgtyp, msgflg);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
```

```
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(retvalue);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(msqid);
  data->arg2 = htonl(msgflg);
  data->arg3 = htonl(retvalue);
  data->op   = htons(183);

  send_manager(data, (void*) msgp, retvalue);
  free(data);

  return retvalue;
}


//msgsnd
int msgsnd(int msqid, const void* msgp, size_t msgsz, int msgflg) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*msgsnd_real)(int, const void*, size_t, int) = NULL;

  if(!msgsnd_real)
    msgsnd_real = dlsym(RTLD_NEXT, "msgsnd");

  retvalue = msgsnd_real(msqid, msgp, msgsz, msgflg);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(msgsz);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(msqid);
  data->arg2 = htonl(msgflg);
  data->arg3 = htonl(retvalue);
  data->op   = htons(182);

  send_manager(data, (void*) msgp, msgsz);
  free(data);

  return retvalue;
}


//pipe
int pipe(int pipefd[2]) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*pipe_real)(int*) = NULL;

  if(!pipe_real)
    pipe_real = dlsym(RTLD_NEXT, "pipe");

  retvalue = pipe_real(pipefd);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
```

```
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(pipefd[0]);
  data->arg2 = htonl(pipefd[1]);
  data->op   = htons(211);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//pipe2
int pipe2(int pipefd[2], int flags) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*pipe2_real)(int*, int) = NULL;

  if(!pipe2_real)
    pipe2_real = dlsym(RTLD_NEXT, "pipe2");

  retvalue = pipe2_real(pipefd, flags);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(pipefd[0]);
  data->arg2 = htonl(pipefd[1]);
  data->arg3 = htonl(flags);
  data->op = htons(212);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//read
ssize_t read(int fd, void* buf, size_t n) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  ssize_t retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static ssize_t (*read_real)(int, void*, size_t) = NULL;

  if(!read_real)
    read_real = dlsym(RTLD_NEXT, "read");

  retvalue = read_real(fd, buf, n);
  fstat(fd, &statbuf);

  data->lwp  = htons(syscall(SYS_gettid));
```

```
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(retvalue);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(fd);
  data->arg2 = htonl(retvalue);
  strcpy(data->path, "");

  if(S_ISSOCK(statbuf.st_mode)) {
    socklen_t len;
    struct sockaddr_storage addr;

    len = sizeof(addr);
    if (getpeername(fd, (struct sockaddr*) &addr, &len) == 0) {
      if (addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->source_port     = s->sin_port;
        data->source_ip       = s->sin_addr;
        data->arg3 = htonl(0);
      } else {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        if(strlen(s->sun_path) < MINPATH)
          strcpy(data->path, "");
        else
          strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }
    if (getsockname(fd, (struct sockaddr*) &addr, &len) == 0) {
      if (addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->dest_port       = s->sin_port;
        data->dest_ip         = s->sin_addr;
        data->arg3 = htonl(0);
      }
    }
    data->op = htons(101);
    data->argPP = protocol_processor(buf, data->source_port, data->dest_port);
  } else if (S_ISFIFO(statbuf.st_mode)) {
    data->op = htons(102);
  } else {
    free(data);
    return retvalue;
  }

  send_manager(data, buf, retvalue);
  free(data);

  return retvalue;
}


//readv
ssize_t readv(int fd, const struct iovec* iov, int iovcnt) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  ssize_t retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static ssize_t (*readv_real)(int, const struct iovec*, int) = NULL;

  if(!readv_real)
    readv_real = dlsym(RTLD_NEXT, "readv");

  retvalue = readv_real(fd, iov, iovcnt);
```

```
  fstat(fd, &statbuf);

  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(retvalue);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(fd);
  data->arg2 = htonl(retvalue);
  data->arg3 = htonl(iovcnt);

  if(S_ISSOCK(statbuf.st_mode)) {
    socklen_t len;
    struct sockaddr_storage addr;

    len = sizeof(addr);
    if(getpeername(fd, (struct sockaddr*) &addr, &len) == 0) {
      if(addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->source_port      = s->sin_port;
        data->source_ip        = s->sin_addr;
      }
    }
    if(getsockname(fd, (struct sockaddr*) &addr, &len) == 0) {
      if(addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->dest_port        = s->sin_port;
        data->dest_ip          = s->sin_addr;
      }
    }
    data->op = htons(231);
  } else if (S_ISFIFO(statbuf.st_mode)) {
    data->op = htons(232);
  } else {
    free(data);
    return retvalue;
  }

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//recv
ssize_t recv(int fd, void* buf, size_t n, int flags) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  ssize_t retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static ssize_t (*recv_real)(int, void*, size_t) = NULL;

  if(!recv_real)
    recv_real = dlsym(RTLD_NEXT, "recv");

  retvalue = recv_real(fd, buf, n);
  fstat(fd, &statbuf);

  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(retvalue);
  data->sec  = htonl(tv.tv_sec);
```

```
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(fd);
  data->arg2 = htonl(retvalue);
  strcpy(data->path, "");

  if(S_ISSOCK(statbuf.st_mode)) {
    socklen_t len;
    struct sockaddr_storage addr;

    len = sizeof(addr);
    if (getpeername(fd, (struct sockaddr*) &addr, &len) == 0) {
      if (addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->source_port     = s->sin_port;
        data->source_ip       = s->sin_addr;
        data->arg3 = htonl(0);
      } else {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        if(strlen(s->sun_path) < MINPATH)
          strcpy(data->path, "");
        else
          strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }
    if (getsockname(fd, (struct sockaddr*) &addr, &len) == 0) {
      if (addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->dest_port       = s->sin_port;
        data->dest_ip         = s->sin_addr;
        data->arg3 = htonl(0);
      }
    }
    data->op = htons(121);
    data->argPP = protocol_processor(buf, data->source_port, data->dest_port);
  } else if (S_ISFIFO(statbuf.st_mode)) {
    data->op = htons(122);
  } else {
    free(data);
    return retvalue;
  }

  send_manager(data, buf, retvalue);
  free(data);

  return retvalue;
}

//semget
int semget(key_t key, int nsems, int semflg) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*semget_real)(key_t, int, int) = NULL;

  if(!semget_real)
    semget_real = dlsym(RTLD_NEXT, "semget");

  retvalue = semget_real(key, nsems, semflg);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
```

```
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(key);
  data->arg2 = htonl(semflg);
  data->arg3 = htonl(retvalue);
  data->op = htons(201);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//semop
int semop(int semid, struct sembuf* sops, size_t nsops) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*semop_real)(int, struct sembuf*, unsigned) = NULL;

  if(!semop_real)
    semop_real = dlsym(RTLD_NEXT, "semop");

  retvalue = semop_real(semid, sops, nsops);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(semid);
  data->arg2 = htons(sops->sem_num);
  data->arg3 = htons(sops->sem_op);
  data->op = htons(202);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//send
ssize_t send(int fd, const void* buf, size_t n, int flags) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  ssize_t retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static ssize_t (*send_real)(int, const void*, size_t) = NULL;

  if(!send_real)
    send_real = dlsym(RTLD_NEXT, "send");

  retvalue = send_real(fd, buf, n);
  fstat(fd, &statbuf);

  data->lwp  = htons(syscall(SYS_gettid));
```

```
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(retvalue);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(fd);
  data->arg2 = htonl(retvalue);
  strcpy(data->path, "");

  if(S_ISSOCK(statbuf.st_mode)) {
    socklen_t len;
    struct sockaddr_storage addr;

    len = sizeof(addr);
    if (getsockname(fd, (struct sockaddr*) &addr, &len) == 0) {
      if(addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->source_port      = s->sin_port;
        data->source_ip        = s->sin_addr;
        data->arg3 = htonl(0);
      } else {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }

    if (getpeername(fd, (struct sockaddr*) &addr, &len) == 0) {
      if (addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->dest_port        = s->sin_port;
        data->dest_ip          = s->sin_addr;
        data->arg3 = htonl(0);
      } else {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        if(strlen(s->sun_path) < MINPATH)
          strcpy(data->path, "");
        else
          strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }

    data->op = htons(131);
    data->argPP = protocol_processor(buf, data->source_port, data->dest_port);
  } else if (S_ISFIFO(statbuf.st_mode)) {
    data->op = htons(132);
  } else {
    free(data);
    return retvalue;
  }

  send_manager(data, buf, retvalue);
  free(data);

  return retvalue;
}


//sendfile
ssize_t sendfile(int out_fd, int in_fd, off_t* offset, size_t count) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  ssize_t retvalue;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static ssize_t (*sendfile_real)(int, int, off_t*, size_t) = NULL;
```

```
  if(!sendfile_real)
    sendfile_real = dlsym(RTLD_NEXT, "sendfile");

  retvalue = sendfile_real(out_fd, in_fd, offset, count);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(retvalue);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(out_fd);
  data->arg2 = htonl(in_fd);
  data->arg3 = htonl(count);
  data->op   = htons(251);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//shmat
void* shmat(int shmid, const void* shmaddr, int shmflg) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  void* retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static void* (*shmat_real)(int, const void*, int) = NULL;

  if(!shmat_real)
    shmat_real = dlsym(RTLD_NEXT, "shmat");

  retvalue = shmat_real(shmid, shmaddr, shmflg);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(shmid);
  data->arg2 = htonl(shmflg);
  data->op   = htons(192);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//shmget
int shmget(key_t key, size_t size, int shmflg) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*shmget_real)(key_t, size_t, int) = NULL;
```

```
  if(!shmget_real)
    shmget_real = dlsym(RTLD_NEXT, "shmget");

  retvalue = shmget_real(key, size, shmflg);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(key);
  data->arg2 = htonl(shmflg);
  data->arg3 = htonl(retvalue);
  data->op   = htons(191);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//socket
int socket(int domain, int type, int protocol) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  int retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static int (*socket_real)(int, int, int) = NULL;

  if(!socket_real)
    socket_real = dlsym(RTLD_NEXT, "socket");

  retvalue = socket_real(domain, type, protocol);

  data->source_port = data->dest_port = 0;
  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(0);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(domain);
  data->arg2 = htonl(type);
  data->arg3 = htonl(retvalue);
  data->op   = htons(141);

  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}


//write
ssize_t write(int fd, const void* buf, size_t n) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  ssize_t retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
```

```
  static ssize_t (*write_real)(int, const void*, size_t) = NULL;

  if(!write_real)
    write_real = dlsym(RTLD_NEXT, "write");

  retvalue = write_real(fd, buf, n);
  fstat(fd, &statbuf);

  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(retvalue);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(fd);
  data->arg2 = htonl(retvalue);
  strcpy(data->path, "");

  if(S_ISSOCK(statbuf.st_mode)) {
    socklen_t len;
    struct sockaddr_storage addr;

    len = sizeof(addr);
    if (getsockname(fd, (struct sockaddr*) &addr, &len) == 0) {
      if(addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->source_port      = s->sin_port;
        data->source_ip        = s->sin_addr;
        data->arg3 = htonl(0);
      } else {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }

    if (getpeername(fd, (struct sockaddr*) &addr, &len) == 0) {
      if (addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->dest_port        = s->sin_port;
        data->dest_ip          = s->sin_addr;
        data->arg3 = htonl(0);
      } else {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        if(strlen(s->sun_path) < MINPATH)
          strcpy(data->path, "");
        else
          strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }

    data->op = htons(111);
    data->argPP = protocol_processor(buf, data->source_port, data->dest_port);
  } else if (S_ISFIFO(statbuf.st_mode)) {
    data->op = htons(112);
  } else {
    free(data);
    return retvalue;
  }

  send_manager(data, buf, retvalue);
  free(data);

  return retvalue;
}
```

```
//writev
ssize_t writev(int fd, const struct iovec* iov, int iovcnt) {
  struct timeval tv;
  gettimeofday(&tv, NULL);

  ssize_t retvalue;
  struct stat statbuf;
  Metadata* data = (Metadata*) malloc(sizeof(Metadata));
  static ssize_t (*writev_real)(int, const struct iovec*, int) = NULL;

  if(!writev_real)
    writev_real = dlsym(RTLD_NEXT, "writev");

  retvalue = writev_real(fd, iov, iovcnt);
  fstat(fd, &statbuf);

  data->lwp  = htons(syscall(SYS_gettid));
  data->pid  = htons(syscall(SYS_getpid));
  data->ppid = htons(syscall(SYS_getppid));
  data->mlen = htonl(retvalue);
  data->sec  = htonl(tv.tv_sec);
  data->usec = htonl(tv.tv_usec);
  data->arg1 = htonl(fd);
  data->arg2 = htonl(retvalue);

  if(S_ISSOCK(statbuf.st_mode)) {
    socklen_t len;
    struct sockaddr_storage addr;

    len = sizeof(addr);
    if(getpeername(fd, (struct sockaddr*) &addr, &len) == 0) {
      if(addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->dest_port        = s->sin_port;
        data->dest_ip          = s->sin_addr;
        data->arg3 = htonl(0);
      } else {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }
    if(getsockname(fd, (struct sockaddr*) &addr, &len) == 0) {
      if(addr.ss_family == AF_INET) {
        struct sockaddr_in* s = (struct sockaddr_in*) &addr;
        data->source_port      = s->sin_port;
        data->source_ip        = s->sin_addr;
        data->arg3 = htonl(0);
      } else {
        struct sockaddr_un* s = (struct sockaddr_un*) &addr;
        strcpy(data->path, s->sun_path);
        data->arg3 = htonl(1);
      }
    }
    data->op = htons(241);
  } else if (S_ISFIFO(statbuf.st_mode)) {
    data->op = htons(242);
  } else {
    free(data);
    return retvalue;
  }
  send_manager(data, (void*) NULL, 0);
  free(data);

  return retvalue;
}
```