Uma arquitetura hierárquica baseada em sistema de arquivos para monitoramento de pacotes de rede no sistema operacional GNU/Linux

Beraldo Costa Leal

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Marco Dimas Gubitoso

São Paulo, agosto de 2013

Uma arquitetura hierárquica baseada em sistema de arquivos para monitoramento de pacotes de rede no sistema operacional GNU/Linux

Esta dissertação de mestrado trata-se da versão original
do aluno Beraldo Costa Leal.

Uma arquitetura hierárquica baseada em sistema de arquivos para monitoramento de pacotes de rede no sistema operacional GNU/Linux

Esta dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa
realizada por Beraldo Costa Leal em 14/10/2013.
O original encontra-se disponível no Instituto de
Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Marco Dimas Gubitoso (orientador) - IME-USP
- Prof. Dr. Daniel Macedo Batista - IME-USP
- Prof. Dr. Jó Ueyama - ICMC-USP

Resumo

Capturar e analisar pacotes de dados que trafegam pelas redes são tarefas essenciais para os administradores de redes. Estas tarefas ajudam na detecção de anomalias nos sistemas e na verificação do estado atual da rede. Existem várias aplicações que desempenham este papel para o sistema operacional *GNU/Linux*. Estes programas também exportam informações para os usuários e outras aplicações de várias maneiras. Entretanto, não exportam estas informações de forma hierárquica.

Esta pesquisa propõe uma arquitetura alternativa aos sistemas atuais. Nossa arquitetura exporta pacotes de dados em uma estrutura hierárquica de arquivos e diretórios. Além disso, por se tratar de uma arquitetura modular, filtros adicionais, desenvolvidos por terceiros, podem ser adicionados ao sistema.

A arquitetura proposta acompanha uma implementação de referência: o sistema de arquivos virtuais *netsfs* (*Network Statistics File System*), que funciona em espaço de núcleo (*kernel space*). A arquitetura e o sistema de arquivos *netsfs*, propostos nesta pesquisa, apresentam um método alternativo para exibir os pacotes de redes. Os resultados mostraram uma aparente melhoria no que diz respeito à vazão da rede.

Palavras-chave: sistema de arquivos, núcleo, *kernel*, *netsfs*, redes, *TCP/IP*, pacotes, sistema operacional, *linux*, *GNU*.

Abstract

Capturing and analyzing data packets flowing across networks are essential tasks for network administrators. These tasks help to detect anomalies in the systems and check the current status of a network. There are software applications for the GNU/Linux operating system which perform such tasks. These tools also export their information to users and other applications in different ways. However, current systems do not export this information in a hierarchical manner.

This research introduces an alternative architecture to current systems. Our architecture exports data packets in a hierarchical structure of directories and files. Furthermore, since this is a modular architecture, additional third-party filters can be developed and loaded into the system.

The proposed architecture comes with a reference implementation: the pseudo file system *netsfs* (Network Statistics File System), in kernel space. The architecture and the pseudo file system *netsfs*, developed in this research, introduce an alternative method to display data packets. Results show an apparent improvement regarding network throughput.
Keywords: file system, kernel, netsfs, network, TCP/IP, packets, operational systems, linux, GNU.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Nos últimos anos a rápida evolução nos padrões de redes de computadores vem tornando o ato de analisar pacotes, com eficiência, um desafio [SWF07]. Nesse contexto, à medida que o hardware é melhorado, o software para gerenciar os dados enviados e recebidos precisa acompanhar essa evolução. Capturar esses pacotes de dados, sem que haja perda de informação, e exibi-los ou armazená-los em disco, precisa ocorrer também em tempo hábil, para que no futuro essa informação seja eventualmente processada.

Atualmente existem algumas ferramentas para a captura e análise dos pacotes de dados que são transmitidos e enviados pela rede. Essa categoria de aplicações recebe o nome de *sniffers*. Os sistemas operacionais modernos utilizam algumas técnicas, como *NAPI* [BHA09], *zero-copy*, [PN08] e *PACKET_SOCKETS* [mpp] para que a entrega do dado para a aplicação (*sniffers*) ocorra da forma mais rápida possível. Algumas destas técnicas utilizadas pelos sistemas operacionais serão explicadas no Capítulo 3 (conceitos).

Para facilitar a análise dos dados, os *sniffers* atuais classificam cada pacote de dados de acordo com os protocolos de redes encontrados. Porém, estas ferramentas não exibem estes dados de uma forma hierárquica, seguindo esta classificação.

Esta pesquisa propõe uma arquitetura alternativa aos sistemas atuais, quanto à forma de exibição dos dados. A arquitetura proposta exporta os pacotes de dados em uma estrutura hierárquica de diretórios e arquivos, exatamente como em um sistema de arquivos e, desta forma, facilita a análise dos dados seja por administradores de redes ou outras aplicações.

É conhecido[DM07] que estas ferramentas, enquanto estão sendo executadas, acrescentam também um gargalo (*bottleneck*) na transferência.

Para testar a arquitetura proposta, uma implementação de referência foi desenvolvida durante este trabalho, o sistema de arquivos **netsfs (Network Statistics File System)**.

Esta implementação de referência foi também comparada com o *tcpdump*[1], principal ferramenta de captura de pacotes nos dias atuais a fim de avaliar o seu desempenho. Nos resultados preliminares, o *netsfs* mostrou uma significativa redução no gargalo gerado na transferência em relação ao *tcpdump*.

1.1 Motivação

No final de 1992 Steven McCanne e Van Jacobson ([MJ93]) apresentaram uma arquitetura para captura e filtragem de pacotes em espaço de usuário, o *BSD Packet Filter - BPF*. Desde então, os principais softwares para captura de pacotes utilizam essa arquitetura. Dentre os mais conhecidos, temos a *libpcap*[2], uma biblioteca portável para captura e análise de pacotes.

Atualmente, os principais softwares para captura de pacotes, incluindo o *tcpdump*, um filtro de pacotes muito utilizado pela comunidade de rede, utilizam a biblioteca *libpcap*, que por sua vez utiliza a sintaxe para filtro de pacotes da arquitetura BPF.

[1] Mesmo site da libpcap
[2] http://www.tcpdump.org/

Do ponto de vista do programador, fazer novos programas que se utilizem dessa biblioteca e dos filtros *BPF* exige um conhecimento sobre a mesma. Do ponto de vista do administrador de redes e/ou sistemas, fazer uma transformação da saída de *sniffers*, como por exemplo, o *tcpdump*, pode ser um trabalho adicional durante a análise dos dados.

Dito isto, é possível melhorar tanto do ponto de vista do administrador de redes quanto do programador, a execução dessas tarefas. Basta que o programador utilize chamadas de sistemas (*syscalls*) já conhecidas e que o administrador de redes utilize ferramentas também já conhecidas por ele, em um ambiente Unix, como ferramentas para exibir o conteúdo de um arquivo, ou para escrever nele.

Todo pacote de rede possui informações sobre quais protocolos (em todos os níveis) ele atua. Este trabalho utiliza estas informações para classificar os pacotes quanto aos protocolos e utiliza-se desta classificação para organizar os pacotes em uma estrutura de diretórios, facilitando assim a análise do tráfego de redes.

Uma das definições do sistema operacional UNIX é que tudo é um arquivo ou um processo [3] e o fato de poder utilizar as mesmas ferramentas para manipular pacotes de redes, assim como arquivos normais, é a principal motivação deste trabalho.

1.2 Objetivos

Este trabalho tem como objetivo propor uma arquitetura para exibição de forma hierárquica dos pacotes de rede que passam pelas interfaces de redes, organizando os pacotes e estatísticas da rede de acordo com as camadas da pilha *TCP/IP* em uma estrutura de diretórios e arquivos.

Durante este trabalho um sistema de arquivos, o **netsfs**, foi desenvolvido para testar e demonstrar esta arquitetura. Trata-se de um sistema de arquivos para o sistema operacional GNU/Linux, que funciona em espaço de *kernel* e implementa a arquitetura proposta neste trabalho.

Dessa forma, o *netsfs* trata-se de um sistema de arquivos virtual para capturar e analisar os pacotes de redes e exibir estes pacotes na forma de arquivos e diretórios.

Com isso, o administrador de redes apenas precisa utilizar ferramentas bem conhecidas para leitura e escritas de arquivos para manipular pacotes de redes. Já o programador não precisará aprender como utilizar bibliotecas como a *libpcap*.

1.3 Contribuições

Uma das contribuições deste trabalho está na geração de uma ferramenta para a captura e análise de pacotes de redes, que poderá ser utilizada por profissionais da área de redes de computadores.

Este trabalho define uma arquitetura simples e estruturada para a análise de pacotes em redes, exibindo os dados em forma hierárquica por meio de arquivos e diretórios.

Esta pesquisa também contribui expondo resultados de experimentos. Estes experimentos mostram o impacto causado por ferramentas de captura de pacotes em hardware comercial (*commodity hardware*). Estes experimentos geraram grande quantidade de pacotes quase que na velocidade de linha (*line rate*) de 10Giga bits por segundo.

1.4 Organização do Trabalho

No Capítulo 3 são apresentados os conceitos fundamentais explorados ao longo deste trabalho e necessários para a implementação do sistema de arquivos de referência proposto. No Capítulo 4 é descrita a principal contribuição deste trabalho, isto é, a arquitetura para captura de pacotes baseada em sistemas de arquivos. No Capítulo 5 é detalhado o desenvolvimento do *netsfs*. E finalmente, no Capítulo 7 são discutidas as conclusões e resultados preliminares deste trabalho, apontando as vantagens da arquitetura proposta.

[3] http://yarchive.net/comp/linux/everything_is_file.html

Capítulo 2

Trabalhos Relacionados

A ideia de representar qualquer tipo de informação em arquivos e diretórios não é nova. Muitos sistemas de arquivos existem com este objetivo, entre os mais conhecidos, pode-se destacar os sistemas de arquivos: *procfs*[1], *sysfs* [Moc05], *GmailFS*[2], *devfs* e sua implementação em espaço de usuário *udev* [KH03], o *demuxfs* [Rea09] entre outros.

Executar ações utilizando chamadas de sistemas normais para leitura e escrita é o grande diferencial destes sistemas de arquivos. Iremos descrever neste capítulo alguns destes sistemas de arquivos para melhor entender a relação com a proposta deste trabalho.

Além dos sistemas de arquivos citados acima, em 1980 *Ken Thompson, Rob Pike, Dave Presotto* e *Phil Winterbottom* iniciaram o sistema operacional *Plan 9* [ea90] na *Bell Labs*, hoje em sua quarta edição (2002).

O sistema de redes do sistema operacional *Plan 9* possui algumas características interessantes, como o fato de exportar via sistemas de arquivos os *sockets* para o espaço de usuário, desta forma o usuário pode escrever e ler dos sockets diretamente nos arquivos [DLP93]. O projeto *Glendix*[3], tenta portar ideias do *Plan 9* para o *kernel Linux*.

2.1 Sistema de arquivos procfs

O *procfs (Proc Filesystem)* é um sistema de arquivos virtual que funciona em memória *RAM* e é montado durante o processo de inicialização do sistema operacional. Trata-se de um sistema de arquivos que exporta por meio da interface */proc/* o estado e informações dos processos em execução em uma estrutura hierárquica de arquivos e diretórios.

Além de expor estas informações por meio de arquivos e diretórios, o *procfs* funciona como um meio de comunicação entre o espaço de usuário e espaço de *kernel*. Algumas aplicações usam esta "interface"para obter status do sistema, como por exemplo a versão *GNU* do comando *ps*.

Sistemas como *Solaris, IRIX, Tru64 UNIX, BSD, Linux, IBM AIX, QNX* e o *Plan 9* suportam o *procfs*.

2.2 Sistema de arquivos devfs

Em sistemas operacionais do tipo *UNIX* um dispositivo possui uma interface para seu *driver* na forma de um arquivo. Utilizando chamadas de sistemas de entrada e saída, usuários e aplicações podem ler e/ou escrever destes arquivos especiais. O *devfs (Device Filesystem)* é o sistema de arquivos responsável por exportar estes arquivos dos dispositivos ao espaço de usuário.

Cada dispositivo possui um ou mais arquivos correspondentes no ponto de montagem do *devfs*, geralmente */dev/*. Por exemplo, o *mouse* possui um arquivo */dev/mice* ou */dev/mouse*, dependendo

[1] https://www.kernel.org/doc/Documentation/filesystems/proc.txt
[2] http://sr71.net/projects/gmailfs/
[3] http://www.glendix.org/

do sistema operacional. Este arquivo exporta informações sobre as posições X e Y do mouse e estado dos botões.

O *devfs* ainda exporta alguns dispositivos virtuais, ou *pseudo-devices*, alguns exemplos: */dev/-null, /dev/zero* e */dev/random*. O primeiro aceita e descarta qualquer entrada sem produzir nenhuma saída, o segundo produz um fluxo contínuo de *NULL bytes* e o último é uma interface para o gerador de números aleatórios do *kernel*.

O *devfs* também é montado durante o processo de inicialização da máquina.

2.3 Sistema de arquivos demuxfs

Em 2009, L. C. Villa Real [Rea09] desenvolveu uma arquitetura para análise de fluxo de dados estruturados aplicada ao sistema brasileiro de TV Digital. A arquitetura proposta por ele pode ser utilizada como ferramenta de depuração do fluxo de áudio e vídeo da TV Digital Brasileira, em forma de sistema de arquivos. O *demuxfs* é a implementação da arquitetura proposta por ele.

O *demuxfs* exporta por meio de arquivos e diretórios informações sobre os fluxos de áudio e vídeo que estão sendo recebidos pelo equipamento decodificador. Além de áudio e vídeo, legendas, logomarcas das emissoras, captura de telas do canal (*snapshot*) e outras informações técnicas são exibidas pelo sistema de arquivos.

Por exemplo para listar os canais (programas) de TV, basta listar um diretório específico:

```
/Mount/DemuxFS] l PAT/Current/Programs
31/12 21:00 rwxrwxrwx          0 0000 -> ../../../NIT/Current
31/12 21:00 rwxrwxrwx          0 0x20 ->  ../../../PMT/0x20/Current
```

Para obter informações sobre determinado fluxo de vídeo, basta também listar outro diretório:

```
/Mount/DemuxFS] l PMT/0x20/Current/VideoStreams/Primary
31/12 21:00 rwxrwxrwx          0 ES
31/12 21:00 rwxrwxrwx          0 PES
31/12 21:00 r-xr-xr-x          0 STREAM_IDENTIFIER
31/12 21:00 r--r--r--          5 elementary_stream_pid
31/12 21:00 r--r--r--          4 es_information_length
31/12 21:00 r--r--r-- 16777215 snapshot.png
31/12 21:00 r--r--r--         79 stream_type_IDENTIFIER
```

Packetized Elementary Streams (PES) são elementos que representam áudio, vídeo e dados (como por exemplo *closed caption*). Trata-se de um *FIFO (First In-First Out)* onde aplicações podem ler dados destes elementos, consumindo-os de forma bem simples, como por exemplo executando: `mplayer PMT/Streams/0x10/PES`.

Inspirado no trabalho de Lucas V. Real, este trabalho usa a mesma ideia: exportar os dados por meio do sistema de arquivos para análise futura. Porém, aqui os dados exibidos são informações dos pacotes que passam pelas interfaces de rede.

Capítulo 3

Conceitos

3.1 Sniffers

Os programas ou *hardwares* capazes de capturar e armazenar o tráfego de uma rede, ou parte dela, são conhecidos como analisador de pacotes, analisador de rede ou *sniffer*. Os pacotes que passam pela rede, estão em formato cru (*raw format*), estes são decodificados, pelo *sniffer*, de acordo com a RFC do protocolo em questão, se necessário.

Os *sniffers* podem capturar e filtrar os pacotes baseados em regras escritas pelo usuário e/ou aplicação, fazendo com que o analisador retorne apenas os pacotes que interessam ao usuário. Estas regras, podem filtrar pacotes por portas, endereços, protocolos nas mais diversas camadas, tamanho do pacote, e até mesmo pelo conteúdo encontrado no pacote, se este não estiver criptografado.

Com um *sniffer* é possível determinar a topologia da rede, a quantidade de tráfego que passa por ela, determinar quais hosts estão gerando maior quantidade de pacotes, e muito mais. Atualmente os analisadores de pacotes / protocolos são essenciais para a administração da rede. Ter um *sniffer* que consiga capturar os pacotes de forma eficiente é essencial para reduzir o gargalo que ele provoca enquanto está em execução.

Entre os principais *sniffers* existentes hoje podemos citar: *ettercap*[1], *wireshark*[2], *ngrep*[3] e o *tcpdump* [4]. Todos eles utilizam a bem conhecida biblioteca *pcap* (*libpcap*). A *libpcap* é uma biblioteca portável para captura e análise de pacotes.

3.2 Sistema de Arquivos

A parte do sistema operacional responsável por armazenar os dados em memória (volátil ou não), acessá-los, modificar seus atributos, atualizá-los, e gerenciar o espaço disponível dessa memória, é o *sistema de arquivos*.

A forma como esses dados são armazenados no dispositivo varia de acordo com o sistema de arquivos. Alguns sistemas de arquivos, gerenciam também atributos para garantir o controle de acesso a determinados arquivos e diretórios a um grupo restrito de usuários, outros por sua vez, criptografam os dados antes de salvá-los.

Um arquivo é apenas uma representação lógica de um bloco de dados que está salvo em um dispositivo, como um disco rígido, por exemplo. O sistema de arquivos mantém esse bloco de dados de forma íntegra no disco. Ele também facilita a exibição desse bloco de dados para o usuário, organizando esses blocos dentro de diretórios.

Os dados são ditos *estruturados*, quando eles necessitam de registros complexos para representar os atributos pertencentes aos dados, atributos como: tamanho, data e hora da última modificação,

[1] http://ettercap.sourceforge.net/
[2] http://www.wireshark.org/
[3] http://ngrep.sourceforge.net/
[4] http://www.tcpdump.org/

permissões, proprietário(s) do dado, e a própria informação. Esses registros no *kernel do Linux* são representados por estrutura de dados (*structs*).

Os sistemas de arquivos podem ser classificados em: a) *físicos*: são os que armazenam os dados em um dispositivo físico (exemplos: iso9660, ext3, fat, entre outros); b) *rede* armazenam dados também em um dispositivo físico, porém utiliza-se da infraestrutura de redes para salvar e manipular o dado remotamente (exemplo: *Network File System - NFS*); c) *pseudos*: estes compartilham a informação baseada em metadados. Por exemplo: podemos ter um sistema de arquivos que salve uma imagem no disco, e que o usuário pode salvar seus meta dados em outro arquivo. Ele automaticamente poderia, salvar esses meta dados na própria figura, evitando que o usuário, necessite de um software para editar os metadados da figura, outro exemplo seria o *PROCFS*, que guarda informações sobre os processos sendo executados na máquina. Mais informações sobre esse último tipo serão detalhadas na seção 3.2.1.

3.2.1 Pseudo sistemas de arquivos

O sistema de arquivos, *netsfs*, que está sendo proposto neste trabalho, é um *pseudo file system* que irá exibir os metadados dos pacotes de rede (origem, destino, protocolos, tamanhos, *flags*, etc..) que passam pelas interfaces de rede em forma de arquivos e diretórios.

Estes tipos de sistemas de arquivos, muitas vezes, na literatura podem ser encontrados, como *pseudo filesystems*, *sistemas de arquivos virtuais*, *sistemas de arquivos de propósito genérico ou especial*.

De acordo com Weinberg [Wei84], os *pseudo file systems* permitem que os dados sejam acessados por meio da interface tradicional de sistemas de arquivos. Estes dados não necessariamente precisam estar localizados em um disco rígido local.

Como exemplo, podemos encontrar o *GmailFS* [5], um sistema de arquivos que exibe informações sobre os emails de um determinado usuário que estão armazenados nos servidores da empresa Google.

Um outro exemplo, é o sistema de arquivos *proc (ProcFS)*, descrito por Erik J. A. K Mouw ([Mou01]). O *ProcFS*, geralmente disponibilizado nos sistemas operacionais em `/proc`, não está associado com um dispositivo de blocos e existe apenas em memória volátil. Ele foi criado com o objetivo de exportar para os programas em espaço de usuário informações sobre os processos sendo executados no *kernel*, mas também é utilizado como um sistema de arquivos para depuração.

A ideia deste sistema de arquivos, proposto por Tom J. Killian ([Kil84]), é implementada em diversos sistemas operacionais, com algumas modificações para exibir informações específicas do sistema operacional. Entre eles, temos: o Solaris ([MM06]), IRIX ([SGI96]), Linux ([Ben05b]) e o Plan 9 ([ea90]).

3.2.2 Sistema de arquivos virtual do Linux

O objetivo de um *Virtual File System - VFS*, originalmente chamado de *Virtual File System Switch*, é de separar o código responsável pelo gerenciamento dos arquivos em dois blocos de código. Um que será uniforme independente do tipo de sistema de arquivos que está sendo acessado, e outro, específico ao sistema de arquivos em questão.

Desta forma, desenvolvedores escrevem aplicações de forma única, sem se preocupar com o sistema de arquivos que de fato está instalado na partição ou disco no qual o acesso está sendo feito. O VFS funciona como uma *API* que o núcleo do sistema operacional oferece para as aplicações em espaço de usuário.

O núcleo Linux, também implementa este conceito. Desta forma, desenvolver um novo sistema de arquivos para o Linux não afeta como as aplicações no espaço de usuário irão manipular os dados contidos neste sistema de arquivos. Veja a Figura 3.1 para entender melhor o papel do *VFS*.

[5] http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html

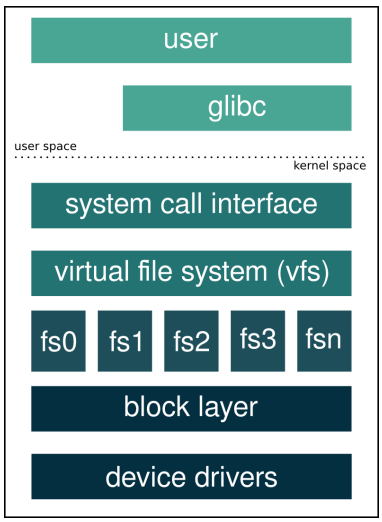


Figura 3.1: *VFS - Virtual File System no núcleo Linux.*

A Figura 3.1 mostra que o *VFS* é uma camada de abstração, do ponto de vista do programador, ou seja, suas chamadas de sistemas são interceptadas pelo *VFS* e este chamará as funções necessárias para as operações dependendo de qual sistema de arquivos ele está lidando.

Do ponto de vista do programador, um `read()` é igual, seja ele para um sistema de arquivos local, ou para um sistema de arquivos em redes.

3.3 Espaço de Kernel e Espaço de Usuário

O gerenciamento de memória é uma das funcionalidades mais importantes que um sistema operacional deve implementar. A manipulação de dados usados por programas e pelo kernel é uma tarefa delicada. Algumas técnicas como como swapping, paginação, memória virtual são implementadas para que o acesso aos dados ocorra com o melhor desempenho possível.

Basicamente o sistema operacional divide a memória virtual em duas regiões. A região de memória dedicada ao núcleo do sistema (*kernel*), o espaço de kernel (*kernel space*) e a região disponibilizada para as aplicações, o espaço de usuário (*user space*). O espaço de kernel como o próprio nome sugere é estritamente reservado para alocar os recursos do núcleo, módulos e drivers de dispositivos, e na maioria dos sistemas operacionais, dados nesta região não são colocados em disco físico (*swapping*). Já o espaço de usuário é o local da memória onde todas as aplicações alocam seus recursos e em contra partida pode ser feito *swapping* para o disco físico.

Como explica, Daniel P. Bovet e Marco C. ([eMC05]), o espaço de memória endereçado para um processo consiste de um conjunto de endereços lineares que o processo pode acessar. Cada processo vê um conjunto diferente de endereços lineares e o endereço usado por um processo não tem relação com o endereço utilizado por outro processo.

Processadores compatíveis a arquitetura *x86* [Int13] possuem modos de proteção, os anéis de proteção (*rings*). O Linux quando executa um trecho de código que corre no espaço de usuário, habilita apenas alguns recursos para este processo, fazendo com que o processo execute no modo menos privilégiado possível. Já um trecho de código que é executado no espaço de kernel (drivers por exemplo) é executado pelo sistema operacional no modo mais privilegiado possível, ou anel 0, podendo assim, este acessar regiões de memória pertencentes a outros programas ou drivers.

3.3.1 Comunicação Entre Espaço de Kernel e Espaço de Usuário

Vários mecanismos de comunicação entre processos (*Inter Process Communication - IPC*) existem para que ocorra a comunicação entre funções que estão sendo executadas no espaço de kernel e espaço de usuário. As aplicações em espaço de usuário podem se comunicar com partes do kernel via

chamadas de sistemas (*system calls*) genéricas (`ioctl`), chamadas de sistemas especificamente desenvolvidas para esta comunicação, sistema de arquivos, onde uma aplicação escreve em um arquivo e o kernel lê do mesmo, ou vice-versa, ou ainda via sockets.

Uma chamada de sistema é uma rotina que está implementada em um nível privilegiado (kernel) e que é chamada por uma aplicação sendo executada em um nível não privilegiado quando deseja acessar algum recurso não disponível diretamente à aplicação.

O código em execução no núcleo Linux pode adicionar entradas ao sistema de arquivos `/proc`, e utilizar estas entradas como comunicação entre o núcleo e as aplicações. Hoje muitas aplicações utilizam-se do `/proc` para obter informações vindas do núcleo (ex: *ps*, *top*, *free*, etc...).

Os sockets do tipo `NETLINK` foram feitos especialmente para estre propósito: comunicação entre espaço de kernel e espaço de usuário, embora possam também ser utilizados para comunicação entre aplicações em espaço de usuário. Trata-se de um mecanismo *IPC*, nos mesmos moldes dos sockets, mas diferente da família de sockets `AF_INET`, a família `AF_NETLINK` não ultrapassa os limites da máquina, ou seja, não existe comunicação máquina-máquina. A RFC 3549 [SNK+03], descreve os sockets do tipo *netlink*.

3.4 Modelo de referência OSI e a arquitetura TCP/IP

Para evitar que as comunicações se tornassem dependentes de tecnologias e empresas, a Organização Internacional para Padronização (*International Organization for Standardization - ISO*), criou um modelo de referência com o objetivo de organizar o processo de comunicação entre dispositivos. Este modelo de referência recebeu o nome de *Open Systems Interconnection model (OSI model)*. Trata-se de um modelo baseado em 7 camadas: aplicação, apresentação, sessão, transporte, rede, enlace e física, onde cada uma destas camadas tem responsabilidades específicas no processo de envio e recebimento de uma mensagem pela rede.

Já o modelo *TCP/IP*, apresentado pela Agência para Pesquisa de Projetos Avançados de Defesa (*Defense Advanced Research Projects Agency - DARPA*), é uma versão simplificada (com apenas 4 camadas) do modelo *OSI*. Ele é definido por um conjunto de protocolos bem definidos como demonstração do funcionamento do modelo *OSI* proposto. As quatro camadas do modelo *TCP/IP* são: aplicação, transporte, internet, e enlace ou rede.

O conjunto de protocolos envolvidos neste modelo, utilizado amplamente até os dias de hoje, é conhecido como a pilha de protocolos *TCP/IP*. Tanto a pilha, quanto o modelo, recebem este nome porque os dois principais protocolos são o *TCP* ([Pro81b]) e o *IP* ([Pro81a]).

Uma breve comparação entre as camadas do modelo *OSI* e o modelo *TCP/IP* encontra-se na Figura 3.2.

Como mostrado na Figura 3.2, o modelo proposto pela *ISO* possui 7 camadas, enquanto que o modelo proposto pela *DARPA* apenas 4.

A arquitetura proposta neste trabalho utiliza a mesma organização adotada pelo núcleo do sistema operacional *Linux*, que mais se parece com a do modelo *DARPA*, um modelo de 4 camadas, mudando apenas os nomes das camadas:

- Camada 1/2 (L 1/2): *Link*;
- Camada 3 (L 3): *Network*;
- Camada 4 (L 4): *Transport*;
- Camada 5 (L 5): *Application*;

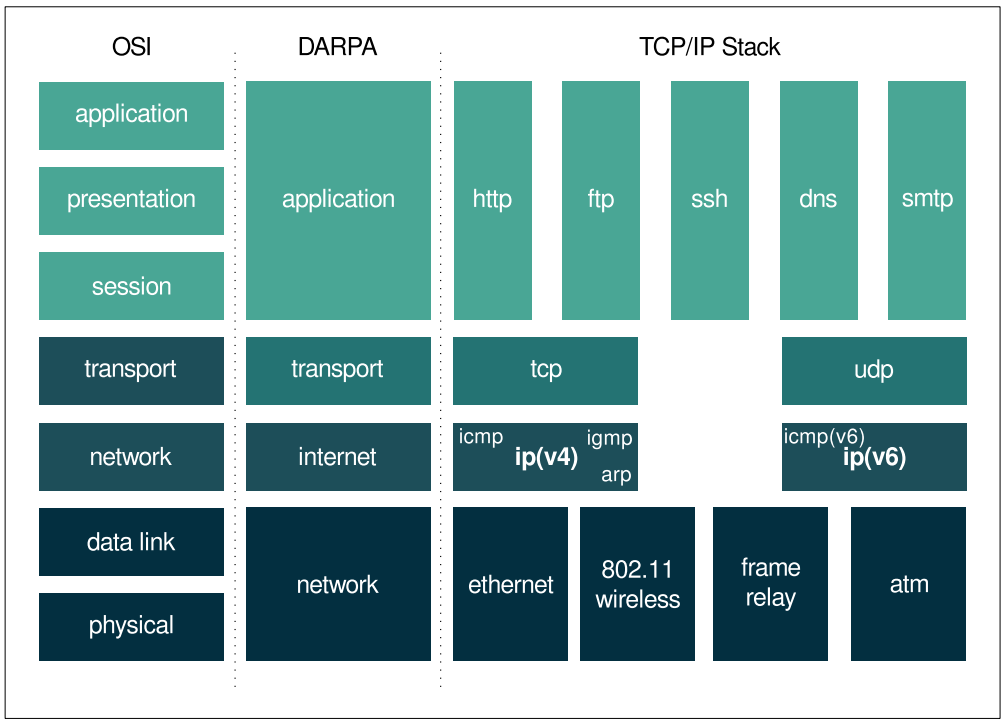


Figura 3.2: *Comparação entre as camadas do modelo OSI e do modelo TCP/IP.*

3.5 Caminho do Pacote de Rede no Linux Kernel

3.5.1 Camada 1/2 - Física / Enlace

Entender o caminho que um pacote percorre no núcleo Linux é fundamental para entender o sistema de arquivos proposto neste trabalho, o *netsfs*. Esta seção irá tratar apenas dos pacotes entrantes no núcleo Linux.

A jornada que um pacote de rede percorre até chegar a aplicação, depende de vários fatores e muitos passos. Melhorias veem sendo feitas em cada camada da arquitetura *TCP/IP* com o objetivo de melhorar a performance da recepção e transmissão destes pacotes[BHA09]. Observe a Figura 3.3 para acompanhar a jornada do pacote de redes dentro do núcleo Linux.

Certamente, todo este processo é um pouco mais complicado. As partes referentes ao roteamento e filtro de pacotes (*netfilter* [6]) foram omitidas nesta explicação. Informações mais detalhadas sobre este processo podem ser encontradas em [Ins02a] [Ins02b] e [Ben05a].

Um dos mais simples e tradicionais métodos para implementar um *driver* de uma interface de rede, consiste em gerar uma interrupção (IRQ) para cada quadro que chega no dispositivo, avisando ao sistema operacional que tem dados para serem lidos. Porém IRQs consomem tempo de CPU, prejudicando o sistema, pois em redes de alto tráfego centenas ou milhares de interrupções por segundo são geradas.

Já a técnica conhecida como *polling* ou *Interrupt Coalescence* [PJD04] pode ser utilizada para evitar muitas interrupções. Neste método, o núcleo vai consultar de tempos em tempos por novos dados no *buffer* da interface de rede. O grande desafio desta técnica é encontrar a frequência correta com que o núcleo irá buscar novas informações. Um *polling* muito frequente pode prejudicar o sistema fazendo com que a CPU fique indisponível para outros processos, dado que ela está sempre ocupada consultando se existem novos quadros no *buffer*. Um *polling* pouco frequente pode criar um estrangulamento na entrega do pacote, dado que os quadros ficarão aguardando para serem processados. Neste caso, alguns quadros podem ser perdidos, caso o *buffer* fique cheio. O núcleo Linux, trabalha por padrão no modo de interrupções (IRQs), mas quando o tráfego de rede atinge um determinado limiar ele muda para o modo de *polling*.

[6] http://www.netfilter.org/

Outro método para avisar ao sistema operacional sobre novos quadros é a nova API(*new API*)[BHA09]. Este método é uma mistura entre interrupções e *polling*. Basicamente ela evita que interrupções muito frequentes aconteçam, gerando apenas novas interrupções quando o *buffer* atinge uma quantidade específica de quadros, entregando-os de uma só vez para o processamento.

O método que é invocado (dentro do *driver* da placa de rede) quando a interrupção é gerada (ou quando o *polling* é executado), é executado com as interrupções desabilitadas. Por esta razão a rotina que processa estes pacotes que estão em buffer, deve terminar o mais rápido possível, para que as interrupções possam ser habilitadas novamente. Basicamente este procedimento: a) aloca uma nova estrutura de dados, chamada `sk_buff`, que do ponto de vista do núcleo Linux é o pacote de redes propriamente dito; b) copia os dados do buffer da placa de redes para esta estrutura recém criada. Esta cópia, dependendo do *driver* da placa de redes, pode ser feita via acesso direto à memória (*Direct Memory Access (DMA)*) como descrito por A. Rubini e J. Corbet [RC05] e c) chama uma função genérica de recebimento, chamada `netif_rx()`. quando esta função retorna, as interrupções são habilitadas novamente, conforme mostrado no item 1 da Figura 3.3.

A função `netif_rx()` basicamente funciona como um concentrador para os quadros que chegam de diferentes interfaces de rede. Em alguns casos (quando a fila está cheia), o quadro é descartado. Em circunstâncias normais a função `__skb_queue_tail()` é chamada e o quadro é enfileirado em um buffer, fora da memória da interface de rede, para um processamento futuro.

A função responsável por desenfileirar o quadro das filas (uma para cada CPU) e enviá-lo para as camadas superiores é a `net_rx_action()` (item 2 e 3 da Figura 3.3). Basicamente esta função desenfileira o quadro e entrega uma cópia deste quadro para cada função encontrada em duas outras listas. Estas listas tem ponteiros para funções que irão tratar protocolos específicos ou genéricos.

Estas funções encontradas nestas listas, são chamadas de ***protocol handlers***. As funções manipuladoras de protocolos (item 4b da Figura 3.3), são registradas no momento do *boot* da máquina, quando um módulo é inicializado ou quando um *socket* é criado.

Por exemplo, para pacotes *IP* existe uma função manipuladora, que irá receber uma cópia do quadro para tratamento *IP*.

3.5.2 Camada 3 - Rede

Se for um pacote *IPv4*, uma função chamada `ip_rcv()` irá manipular este datagrama *IP* (item 4a da Figura 3.3). Está função irá executar as checagens de soma (*checksum*), verificar os campos do cabeçalho *IP*, tamanho do datagrama, etc. Se o datagrama estiver correto a função `ip_rcv_finish()` será chamada (item 6 da Figura 3.3).

Esta função basicamente implementa as questões relacionadas ao roteamento, decidindo se o pacote irá será encaminhado para outra interface de rede (item 7b) ou se irá prosseguir e ser entregue para alguma aplicação no espaço de usuário (item 7a). Algumas outras funções são executadas para realizar as demais tarefas relacionadas ao *IP*, como a remontagem do datagrama caso ele esteja fragmentado.

Exceto os casos em que o pacote é processado pelo próprio núcleo (*ICMP* e *IGMP*) o pacote continua seu caminho através das camadas superiores e a estrutura `sk_buff` vai sendo preenchida com as informações do pacote a medida que passa pelas funções.

Neste momento, novamente as filas contendo as funções manipuladoras de protocolos são percorridas. Se, por exemplo, trata-se de um pacote *TCP*, uma função específica para tratar pacotes *TCP* irá receber uma cópia do segmento *TCP* (item 9 da Figura 3.3).

3.5.3 Camada 4 - Transporte

Na camada de transporte, se o pacote for do tipo *TCP*, ele será chamado de segmento, se ele for do tipo *UDP*, ele continuará sendo chamado de datagrama. Está fora do escopo deste trabalho explicar detalhadamente o tratamento de um segmento *TCP* ou de um datagrama *UDP*.

Basicamente, nesta etapa, a função `tcp_v4_rcv()` (item 9 da Figura 3.3) verifica a integridade dos cabeçalhos TCP e então localiza se existe um *socket* em estado de *listen*, baseado no campo

porta de destino. No caso de não existir um *socket* aberto, uma mensagem *ICMP* será gerada para ser enviada ao remetente. Caso exista, a função `tcp_v4_do_rcv()` será chamada passando a estrutura `sk_buff` e o *socket* encontrado como parâmetros. Esta função irá executar diferentes ações dependendo do estado do *socket*.

O ponto mais importante para se observar na recepção de um pacote de redes no núcleo Linux é a função `net_rx_action()` (item 3 da Figura 3.3), como mostrado na seção 3.5.1. Observe na Figura 3.3 que ela pode executar uma função para manipular uma cópia do pacote dependendo do tipo do protocolo ou até mesmo todos os tipos de protocolos. Este é um dos métodos mais utilizados para a captura de pacotes de redes, método este, que será utilizado para a implementação do sistema de arquivos proposto por este trabalho.

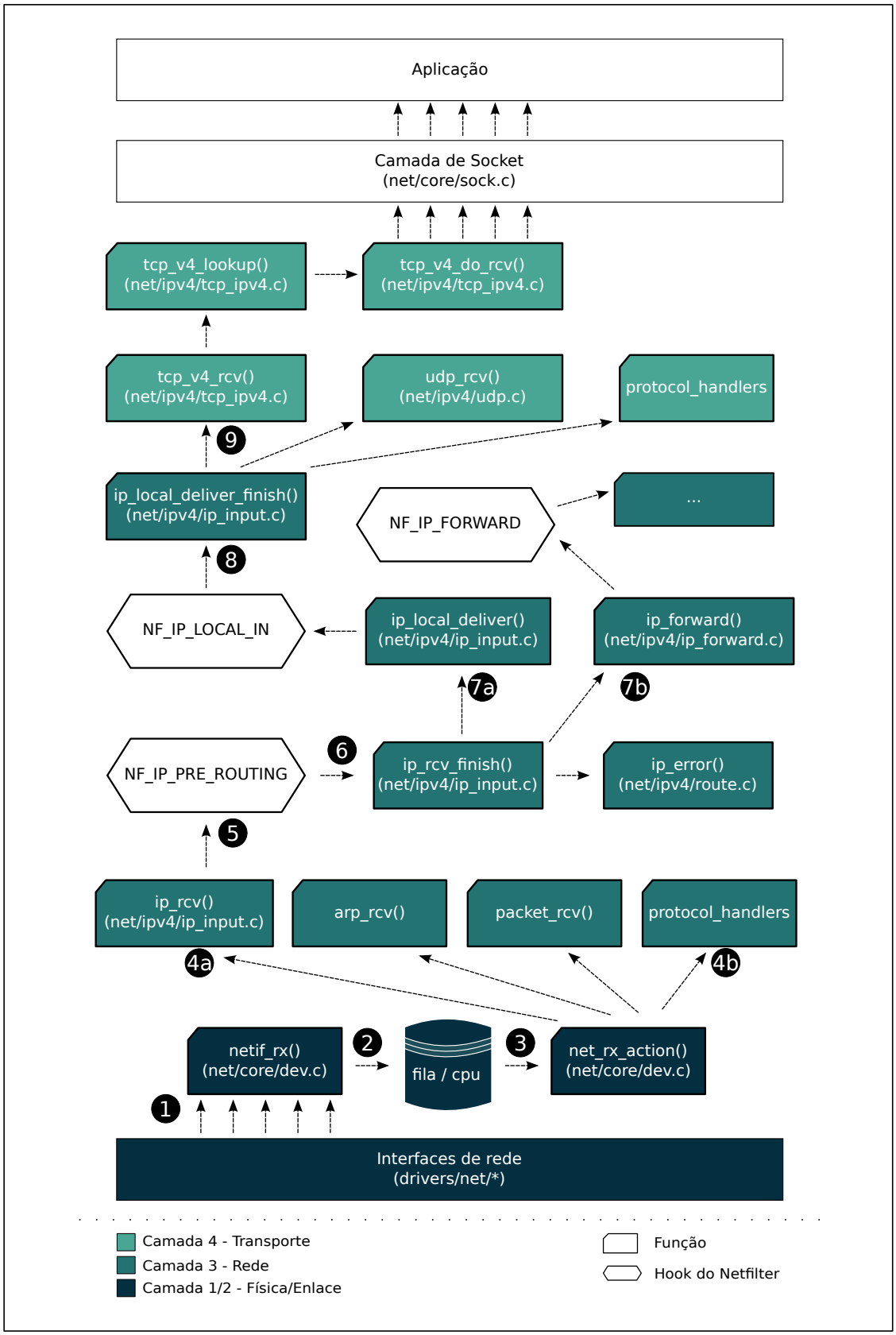


Figura 3.3: *Jornada do pacote de redes dentro do núcleo Linux*

Capítulo 4

Uma arquitetura para monitoramento de pacotes de rede

Este capítulo apresenta a principal contribuição deste trabalho: uma arquitetura onde a exibição dos meta dados extraídos dos pacotes ocorre em um sistema de arquivos, para facilitar a leitura e entendimento por parte do usuário.

A arquitetura proposta também é modular, o que permite que novos filtros ou novos contadores possam ser acoplados à mesma. Essa arquitetura pode contar com mecanismo de controle de saída. Esse mecanismo é responsável por determinar o nível de verbosidade da saída dos módulos e/ou do processamento padrão, exibindo assim, mais ou menos informações no sistema de arquivos.

4.1 Elementos da arquitetura

A arquitetura proposta envolve 5 componentes básicos: a) captura dos dados vindos das interfaces de rede – esse componente funciona como um concentrador, que recebe os pacotes de redes de múltiplas interfaces de redes; b) o processamento padrão é o componente responsável pelos contadores de pacotes, contadores de bytes, contadores de protocolos e coleta de básicas de meta dados; c) os módulos são registrados na arquitetura informando que tipo de dados interessam aos módulos, e esses farão suas tarefas de forma independente do processamento padrão; d) controle de saída, que permite ao usuário controlar a verbosidade da saída e d) o sistema de arquivos, responsável por expor ao usuário o que o processamento padrão processou e o que os módulos processaram nas etapas anteriores.

A Figura 4.1 mostra os componentes dessa arquitetura, baseada na proposta de L. C. Real [Rea09] para o contexto da TV digital brasileira.

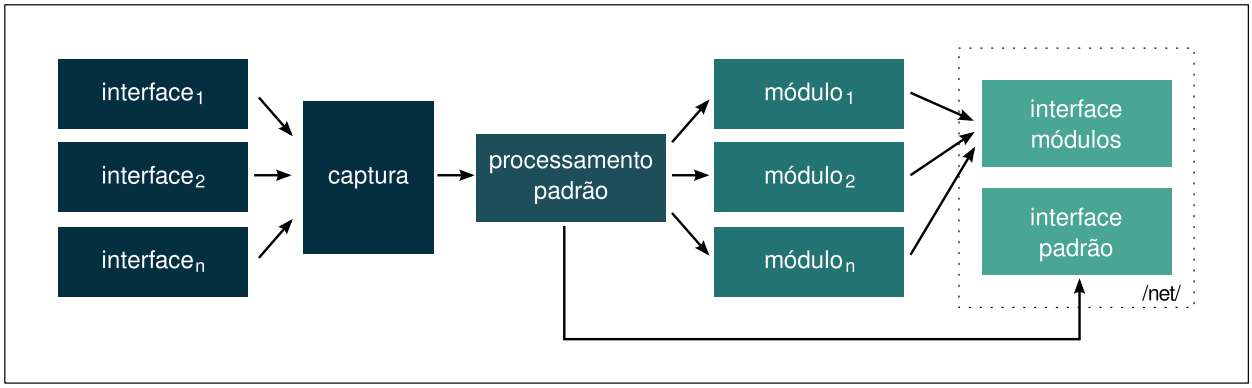


Figura 4.1: *Elementos da arquitetura.*

As interfaces mostradas na Figura 4.1 são as interfaces de redes por onde os pacotes passam. Um concentrador é responsável pela captura de todos os pacotes vindo destas interfaces, este con-

centrador pode ser implementado utilizando os *protocols handlers*. Estes por sua vez, analisam os pacotes e expõe os dados na interface padrão, onde esta interface trata-se de uma estrutura de arquivos e diretórios, por exemplo: */net/*.

A arquitetura é modular, permitindo que novos módulos sejam acoplados ao sistema, para tratamento específico de pacotes. Por exemplo, pode-se ter um módulo para tratamento de pacotes de *VoIP* onde ele não está preocupado com informações de baixo nível do pacote, mas sim sobre o áudio da conversa, salvando este áudio no sistema de arquivos após analisar todos os pacotes da conexão.

4.2 Exposição dos dados estruturados em sistema de arquivos

O *sistema de arquivos* é responsável por expor os meta dados aos usuários. A forma como esses dados serão disponibilizados é de fundamental importância para uma boa usabilidade do sistema.

O que é proposto nessa arquitetura é uma exibição de forma hierárquica, onde em cada diretório do sistema de arquivos representa um nível da pilha *TCP/IP*, o que facilita a busca por contadores e meta dados. Supondo que o sistema de arquivos foi montado e disponibilizado pelo sistema em `/net`, a Figura 4.2 abaixo apresenta uma proposta de sistema de arquivos para essa nova arquitetura.

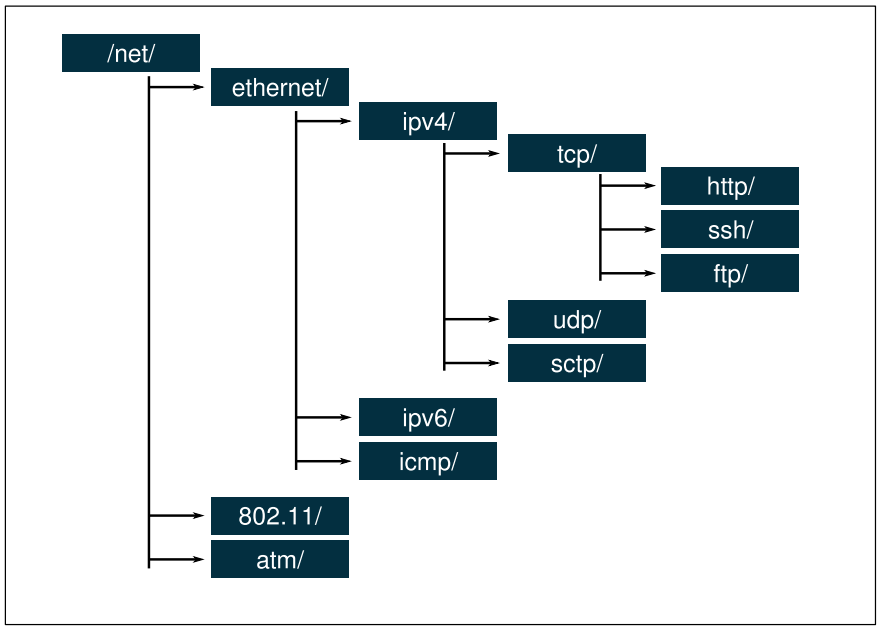


Figura 4.2: *Exposição dos dados estruturados de forma hierárquica em um sistema de arquivos.*

Além dessa estrutura, cada nível deve conter ao menos dois arquivos especiais: `stats` e `stream`. O primeiro contém contadores dos pacotes e *bytes* do nível correspondente. O segundo é um arquivo para o fluxo de dados que passa no momento da leitura do arquivo.

Com essa estrutura, a leitura dos meta dados dos pacotes de redes se dará de forma natural, uma vez que o usuário precisará apenas navegar entre os diretórios para coletar os dados da camada que necessitar. Ao utilizar ferramentas disponíveis no sistema operacional, como `cat` e `tail`, ele poderá ler os contadores ou o fluxo de dados em tempo real.

4.2.1 Buffer circular para o arquivo stream

Quando novos pacotes chegam nas interfaces de rede, o concentrador da arquitetura é responsável por armazenar estes dados nos respectivos *buffers*. Após uma análise para descobrir quais protocolos existem no pacote, o mesmo é copiado para um *buffer* correspondente para cada camada.

Desta forma, um pacote de rede que contém os protocolos: *ethernet*, *ip*, *tcp* e *http* é copiado quatro vezes, um para cada *buffer*. Sendo assim, o componente da arquitetura responsável pela

exibição dos dados fará a leitura do *buffer* correspondente ao diretório em que o usuário está navegando.

Este *buffer* trata-se de um *buffer circular simples*. O início e o fim do *buffer* é mostrado na Figura 4.3.

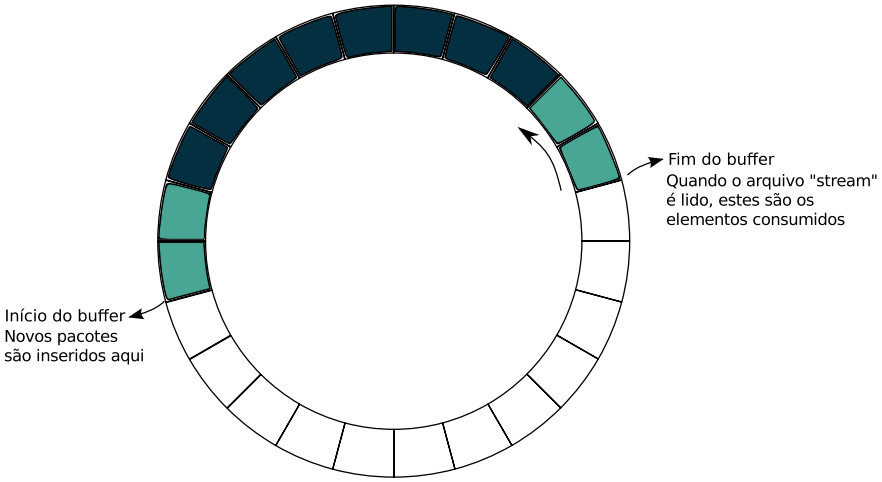


Figura 4.3: *Buffer circular que o arquivo "stream"está associado*

Os arquivos *stream* estão associados a estes *buffers*.

4.2.2 Controle de saída

Como mencionado anteriormente, os pacotes que passam pelas interfaces são exibidos no arquivo `stream`, que se encontra em cada nível dos diretórios. A quantidade de informação que será exibida tem que ser personalizável pelo usuários por algum canal de comunicação, por exemplo, um arquivo de controle, *debug*.

A seguir dois exemplos: no primeiro, a Figura 4.4 mostra um possível exemplo de saída para o arquivo `stream`, onde pouca verbosidade está configurada; já na Figura 4.5 com mais informações, como por exemplo, informações de cabeçalhos adicionais.

```
11:33:39.827722 IP 192.168.1.102 > 74.125.234.52: ICMP echo request, length 64
11:33:39.841209 IP 74.125.234.52 > 192.168.1.102: ICMP echo reply, length 64
11:33:40.829159 IP 192.168.1.102 > 74.125.234.52: ICMP echo request, length 64
```

Figura 4.4: *Possível exemplo de saída com pouca verbosidade.*

Esses exemplos de saída são idênticos às saídas do comando *tcpdump*, que são informações úteis para a análise do tráfego da rede. Como será apresentado na Seção 4.3, dependendo do módulo que estiver acoplado ao sistema, essa saída pode ser, em outro arquivo, uma informação previamente tratada, como, por exemplo, um arquivo de aúdio.

4.2.3 Filtros

A arquitetura, além de permitir o nível de verbosidade, deve permitir também que filtros sejam aplicados aos dados em cada um dos níveis. Assim, o usuário poderá, por exemplo, escolher quais pacotes interessam e quais não interessam.

Predende-se manter a sintaxe desses filtros igual ao filtros *BPF*. Pode-se, por exemplo, ter filtros que apenas exibam pacotes que tenham endereço de origem a um determinado *IP*. Isso evitará que o arquivo, que contém o fluxo de pacotes atual, exiba informações que não interessam, como pacotes desnecessários, por exemplo.

Esses filtros precisam ser configurados em tempo real, por meio do arquivo de contrle.

```
11:46:32.987190 IP 192.168.1.102 > 74.125.234.49: ICMP echo request, length 64
0x0000:  4500 0054 0000 4000 4001 43ec c0a8 0166  E..T..@.@.C....f
0x0010:  4a7d ea31 0800 b4ae 3b61 0003 c8ba 5c4f  J}.1....;a....\O
0x0020:  0000 0000 1510 0f00 0000 0000 1011 1213  ................
0x0030:  1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  .............!"#
0x0040:  2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
0x0050:  3435 3637 4567
1e1f1:46:32.998815 IP 74.125.234.49 > 192.168.1.102: ICMP echo reply, length 64
0x0000:  4500 0054 56ba 0000 3901 3432 4a7d ea31  E..TV...9.42J}.1
0x0010:  c0a8 0166 0000 bcae 3b61 0003 c8ba 5c4f  ...f....;a....\O
0x0020:  0000 0000 1510 0f00 0000 0000 1011 1213  ................
0x0030:  1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  .............!"#
0x0040:  2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
0x0050:  3435 3637 4567
11:46:33.987368 IP 192.168.1.102 > 74.125.234.49: ICMP echo request, length 64
0x0000:  4500 0054 0000 4000 4001 43ec c0a8 0166  E..T..@.@.C....f
0x0010:  4a7d ea31 0800 ffac 3b61 0004 c9ba 5c4f  J}.1....;a....\O
0x0020:  0000 0000 c910 0f00 0000 0000 1011 1213  ................
0x0030:  1155 1617 1819 1a1b 1c1d 1e1f 2021 2223  .............!"#
0x0040:  2425 2425 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
0x0050:  3435 3637
```

Figura 4.5: *Possível exemplo de saída com muita verbosidade.*

4.3 Módulos

Como exibido na Figura 4.1, um dos componentes da arquitetura são os módulos. Módulos permitem que, não apenas os dados puros sejam ser exibidos, mas também informações previamente tratadas que possam ser acessadas por alguma aplicação. Os módulos permitem que novos filtros sejam desenvolvidos, o que torna a arquitetura flexível.

Esses módulos, basicamente, irão se registrar junto ao núcleo do sistema (processamento padrão), avisando que tipo de pacotes os interessam. Durante o processamento padrão, quando um pacote que interesse a esse módulo (baseado no protocolo) chegar pelas interfaces de rede, esse pacote deverá ser copiado para que o módulo faça o tratamento devido. É de total responsabilidade do módulo a liberação desta cópia do pacote, que por sua vez após utilização pelo módulo, não faz mais sentido ao sistema.

Os módulos podem ser úteis para tratamento especial de algum tipo de pacote, ou para alguma aplicação específica. Por exemplo, pode ser desenvolvido um módulo que monitore apenas pacotes de ***Voz sobre IP (VoIP)***, e que esse módulo salve os arquivos de áudio (fazendo a remontagem do pacote) em um determinado diretório.

Outro exemplo, seria um módulo específico para ***detectar intrusos baseado em assinaturas de pacotes maliciosos***. Esse módulo poderia escrever alertas em um arquivo, que uma aplicação estivesse monitorando. Ainda, seria possível bloquear os pacotes que casessem com essas assinaturas.

Por fim, também deve-se permitir que módulos exibam dados estatísticos mais detalhados do que aqueles exibidos pelo processamento padrão.

Cada módulo pode acrescentar novas funcionalidades ao sistema, como opções de filtros que são reconhecidas, formato de saída de dados, suporte a novos protocolos. O usuário poderá escolher quais módulos ele deseja habilitar ou não, uma vez que isso terá um impacto no desempenho do sistema.

Capítulo 5

Implementação de referência - netsfs

Este Capítulo apresenta as características do sistema de arquivo de referência, proposto aqui, o *netsfs*, do inglês *Network Statistics File System*. Trata-se de um sistema de arquivos que segue a arquitetura. Sua implementação é baseada nos conceitos apresentados na seção 3.2 deste trabalho.

O *netsfs*, está em versão de desenvolvimento e disponível no *github*[1], disponibilizado sob a licença *GPL v2*. Ele está sendo desenvolvido com base na versão 3.10.0 do kernel Linux.

5.1 Funcionamento do netsfs

O netsfs, é um sistema de arquivos volátil, guardando seus dados (contadores e estrutura de diretórios) em memória *RAM*. Ele exibe os dados desde o momento em que o sistema de arquivos foi iniciado (montado). Ao reiniciar o computador, ou desmontar o sistema de arquivos, as informações serão perdidas. Seu funcionamento é semelhante ao sistema de arquivos *procfs* [Kil84].

O netsfs é um sistema de arquivos que corre em espaço de kernel, em forma de módulo, e utiliza-se do sistema de arquivos virtual fornecido pelo núcleo Linux, conforme mostrado na seção 3.2.1. Porém, nada impede que o mesmo seja implementado em espaço de usuário, utilizando a *lib fuse* [SZE09], uma biblioteca com a qual é possível implementar um sistema de arquivos completo em espaço de usuário.

Ele registra uma função *protocol handler*, conforme visto na seção 3.5.1, avisando ao kernel que quer receber uma cópia de todos os pacotes que passam pelas interfaces de rede. Esta função, por sua vez, faz o tratamento necessário obtendo informações dos pacotes e exibindo no sistema de arquivos.

O sistema de arquivos netsfs, depois de compilado e instalado, pode ser facilmente disponibilizado para o usuário com os seguintes comandos:

```
# modprobe netsfs.ko
# mkdir /net
# mount -t netsfs none /net
```

Figura 5.1: *Comandos para carregar e montar o sistema de arquivos netsfs.*

Como mostrado na Figura 5.1, o primeiro comando, carrega o módulo previamente compilado junto ao kernel do Linux, o segundo comando cria um diretório vazio chamado */net*, e o último comando monta o sistema de arquivos neste diretório recém criado. Como se trata de um *pseudo* sistema de arquivos, este não está associado a nenhum dispositivo.

Neste momento, todos os pacotes que passam pelas interfaces de redes estão sendo processados pelo netsfs, que por sua vez, está montando a estrutura de diretório, baseada nos protocolos, em tempo real, a medida que os pacotes são interceptados.

A Figura 5.2 mostra um exemplo da listagem do diretório */net/ethernet/ipv4*:

[1] https://github.com/beraldoleal/netsfs

```
# ls -la /net/ethernet/ipv4/
total 24
drwxr-xr-x 4 root root 4096 Mar 11 14:54 .
drwxr-xr-x 3 root root 4096 Mar 11 14:54 ..
-rw-r--r-- 1 root root    2 Mar 11 14:54 verbose
-rw-r--r-- 1 root root   20 Mar 11 14:53 stats
-rw-r--r-- 1 root root    0 Mar 11 14:53 stream
drwxr-xr-x 2 root root 4096 Mar 11 14:53 tcp
drwxr-xr-x 2 root root 4096 Mar 11 14:53 udp
```

Figura 5.2: *Exemplo de listagem do diretório ipv4 depois de montado o netsfs.*

5.1.1 Implementação do controle de saída

No netsfs, o controle de saída é implementado por meio de um arquivo de controle, chamado *verbose*. Este arquivo encontra-se em cada diretório e aceita que seja escrito um inteiro maior ou igual a 0. Quanto maior o número maior será o nível de verbosidade exibida no arquivo *stream*.

O usuário pode escrever neste arquivo o valor de verbosidade que ele deseja. A Figura 5.3 exibe um exemplo de como o usuário poderia efetuar a escrita.

```
# echo 9 > /net/ethernet/ipv4/verbose
```

Figura 5.3: *Exemplo de comando que aumenta a verbosidade para 9.*

5.1.2 Implementação do arquivo stream

No netsfs, o arquivo que contém o fluxo de dados que passam pelas interfaces, é um *FIFO - (First in first out)* [IEE01] em forma de *buffer circular*, conforme descrito em 4.2.1. Um ou mais processos podem abrir o *FIFO* para leitura.

Um processo consumidor pode ainda abrir o *FIFO* para leitura de modo *não bloqueante*, ou seja, ele retorna sucesso, mesmo que o processo consumidor não tenha no *FIFO*.

Desta forma, não precisamos armazenar em disco nem em memória os dados que chegam. Apenas quando o *FIFO* estiver com pelo menos um processo lendo, seus dados serão passados.

O usuário poderá fazer um tail, como mostra a Figura 5.4 abaixo:

```
# tail -f /net/ethernet/icmp/stream
11:33:39.827722 IP 192.168.1.102 > 74.125.234.52: ICMP echo request, length
11:33:39.841209 IP 74.125.234.52 > 192.168.1.102: ICMP echo reply, length 64
11:33:40.829159 IP 192.168.1.102 > 74.125.234.52: ICMP echo request, length
```

Figura 5.4: *Exemplo do comando de leitura do FIFO.*

Atualmente o sistema de arquivo suporta apenas *cat* para leitura.

5.1.3 Módulos no netsfs

O próprio netsfs é um módulo plugável do kernel do Linux, porém ele exportará alguns símbolos, por meio da função `EXPORT_SYMBOL()` [Rus00], tornando assim possível que outros módulos desenvolvidos por terceiros possam utilizar-se de funções encontradas no netsfs.

Sendo assim, um outro módulo poderá se registrar no netsfs, por meio de uma função, avisando que quando um pacote de um determinado protocolo passar por uma interface de rede, este deve ser copiado para uma função contida neste segundo módulo.

O *netfilter* [JE11], arquitetura dentro do núcleo do Linux responsável pelas regras de *firewall*, funciona do mesmo jeito, neste aspecto. Ele permite que novos módulos sejam desenvolvidos e novos

filtros adicionados ao *netfilter*. Por exemplo, existe o módulo `nf_log` que permite que regras de log sejam adicionadas ao *iptables*.

Capítulo 6

Análise comparativa

Neste capítulo apresentamos uma análise comparando o impacto causado pelo sistema de arquivos *netsfs* na vazão da rede (*throughput*) assim como no tempo utilizado de CPU durante sua execução.

Atualmente, por se tratar de uma implementação de referência básica, o *netsfs* é bastante limitado em seus recursos comparado aos programas atuais com o mesmo propósito: captura e análise de pacotes. Entretanto, uma análise e classificação básica dos pacotes pode ser feita.

De qualquer forma, neste capítulo também apresentamos uma análise comparativa do *netsfs* e a principal ferramenta para captura de pacotes no sistema operacional GNU/Linux, o *tcpdump*.

O *tcpdump* utiliza a *libpcap* para capturar e executar os filtros de pacotes. Esta biblioteca, como mencionado anteriormente, é a biblioteca padrão em se tratando de análise e captura de pacotes.

Esta comparação entre o *tcpdump* e a ferramenta proposta neste trabalho é fundamental para entender melhor o comportamento de uma aplicação deste tipo funcionando em espaço de *kernel*.

6.1 Ambiente dos experimentos

Os experimentos foram executados em um ambiente controlado. As máquinas utilizadas foram instaladas para este propósito a fim de evitar interferência de outros programas em execução durante os experimentos [Jai91].

Duas máquinas (servidores) foram utilizados para estes experimentos, uma máquina (máquina *A*) foi a responsável por gerar o tráfego de rede para a máquina *B*, que continha o software fazendo a captura dos pacotes. As duas máquinas estavam ligadas por meio de um cabo de par trançado categoria 6a, que atinge um limite teórico de *10 Gigabits* por segundo [STA01], ambas utilizando placas de redes de *10 Gbps*.

A máquina *A* gerou tráfego de rede por meio da ferramenta *Iperf*[1]. Esta ferramenta pode gerar fluxos tanto *TCP* quanto *UDP*, e medir algumas métricas deste tráfego. Este programa, escrito em *C++*, foi desenvolvido pelo *Distributed Applications Support Team (DAST)* no *National Laboratory for Applied Network Research (NLANR)*.

Na Tabela 6.1, temos as configurações de *hardware* utilizadas nas duas máquinas em que os experimentos foram executados:

[1] http://iperf.sourceforge.net/

Modelo CPU	Intel(R) Xeon(R) CPU E5-2650
Frequência	2 Ghz
Quantidade de Núcelos	32
Threads por núcleo	2
Arquitetura	x86_64
Memória RAM	15995 MB
Interface de rede	Intel Corporation 82599EB 10 Gigabit TN Network
Linux Kernel	3.8.0-19-generic
Distribuição	Ubuntu Server 13.04

Tabela 6.1: *Configuração das máquinas em que os experimentos foram executados*

6.2 Coleta dos dados

Uma terceira máquina (*máquina C*) foi utilizada para fazer a coleta dos dados. Esta máquina funcionou de forma passiva no ambiente dos experimentos, executando coletas a cada *quinze segundos* na *máquina B* por meio do protocolo *SNMP (Simple Network Management Protocol)* [Sta98].

A coleta dos dados não afetava de forma direta os experimentos em execução, pois uma outra interface de rede foi utilizada para fazê-la. Sendo assim a *máquina B* possuía duas interfaces de rede, uma para responder as requisições *SNMP* e outra, de *10 Gbps*, para os experimento com o *Iperf*.

Cada experimento ficou em execução por 10 minutos e foi executado 10 vezes (40 amostras), com um *reboot* entre cada experimento. Estes valores foram escolhidos de forma empírica.

6.3 Impacto do tcpdump no sistema

Este primeiro experimento mostra o impacto do *tcpdump* em um sistema inicialmente ocioso (*idle*). Com o objetivo de igualar ao máximo as comparações, algumas opções foram passadas para o *tcpdump* para que ele se comporte o mais parecido possível com o *netsfs*.

Opções passadas:

- -K, para não verificar *checksum* dos cabeçalhos IP, TCP ou UDP.
- -n, para não converter endereços e portas para nomes, exibindo-os em forma numérica.
- -O, não executa o optimizador de filtro de correspondência dos pacotes (*packet-matching*).

6.3.1 tcpdump com filtro de camada 2

Inicialmente o sistema foi medido sem nenhum software em execução, em estado ocioso, (10 medições de 10 minutos). Logo em seguida o tcpdump foi executado fazendo apenas filtro de camada 2, ou seja, sem filtro de *IP* de origem ou destino e portas de origem e destino. Desta forma o *tcpdump* capturou os pacotes que chegaram no dispositivo de rede à 10 Gbps, no *Linux* reconhecido com o nome *p802p*.

Comando executado para a captura dos pacotes na *máquina B*:

```
$ tcpdump -n -K -O -i p802p > /dev/null
```

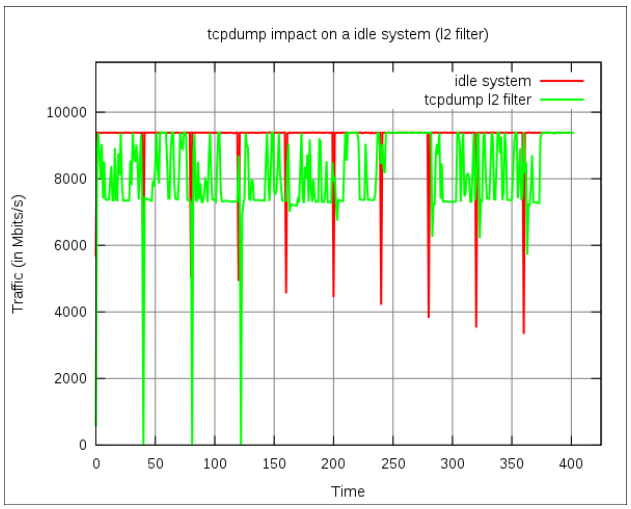


Figura 6.1: *Tráfego de rede normal versus tráfego de rede com o tcpdump*

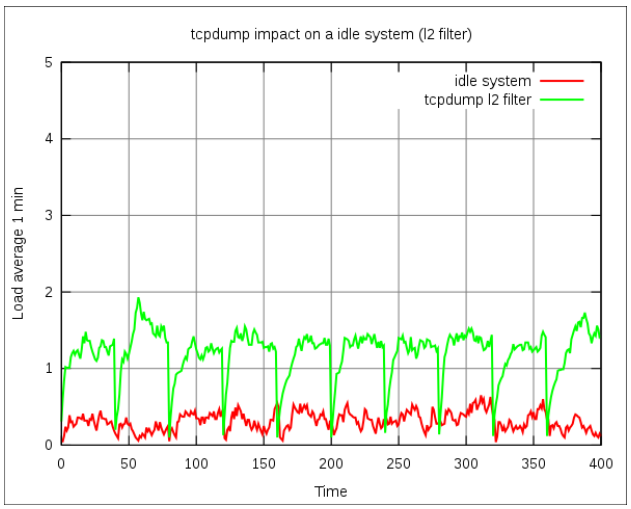


Figura 6.2: *Carga de CPU normal versus carga de CPU com o tcpdump*

Observando as Figuras 6.1 e 6.2, pode-se notar uma queda na vazão da rede e um aumento na carga de CPU. Os valores abaixo de 6.000 Mbps na 6.1 acontecem por causa do *reboot* que a máquina sofreu entre cada um dos testes.

A carga de CPU (*load average*) medida durante este e todos os outros testes é uma média do último minuto, representando a quantidade média de processos que estavam em espera para serem executados [Gun10].

A tabela 6.2 mostra um resumo dos dados coletados:

	Sistema ocioso	tcpdump (L2)
Tráfego de rede em GBps		
Mínimo	3.3365920	0.5596651
Máximo	9.3894851	9.3882826
Média	9.2610751	8.1576127
Carga média de CPU (1 minuto)		
Mínimo	0.0400	0.1000
Máximo	0.6500	1.9300
Média	0.3108	1.2118

Tabela 6.2: *Valores coletados na comparação do sistema ocioso e o tcpdump com filtro de camada 2*

6.3.2 tcpdump com filtro de camada 4

Neste experimento pretende-se observar o comportamento do *tcpdump* utilizando um filtro mais complexo de pacotes, fazendo com que o *tcpdump* analise os cabeçalhos da camada 4 (porta de destino).

Para fazer esta análise, o *tcpdump* precisa:

- Determinar o protocolo de camada 3, analisando os cabeçalhos da camada 2;
- Determinar o protocolo da camada 4 (TCP, UDP), analisando os cabeçalhos da camada 3;
- Fazer combinar apenas de pacotes TCP;
- Fazer combinar de porta, comparando a porta passada na linha de comando com o campo da porta de destino no cabeçalho da camada 4.

Neste cenário a mesma metodologia do primeiro cenário foi adotada, 10 medições de 10 minutos com intervalo de coleta de 15 segundos.

O *Iperf*, utilizado nos experimentos, utiliza a porta 5001 por padrão, por esta razão a linha de comando utilizada na *máquina B*, foi:

```
$ tcpdump -n -K -O -i p802p 'tcp and port 5001' > /dev/null
```

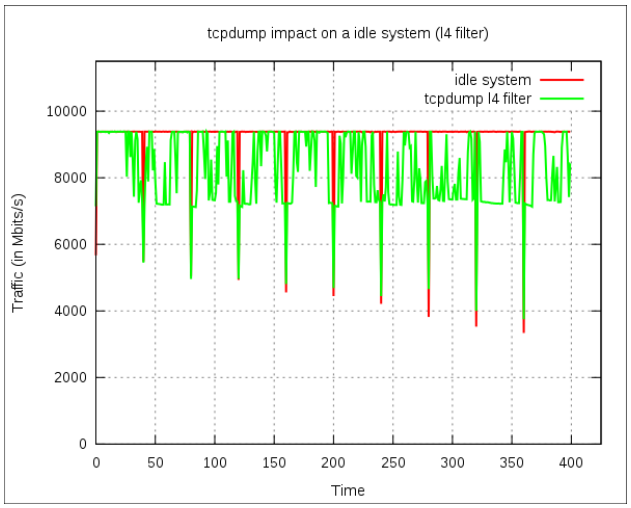


Figura 6.3: *Tráfego de rede normal versus tráfego de rede com o tcpdump*

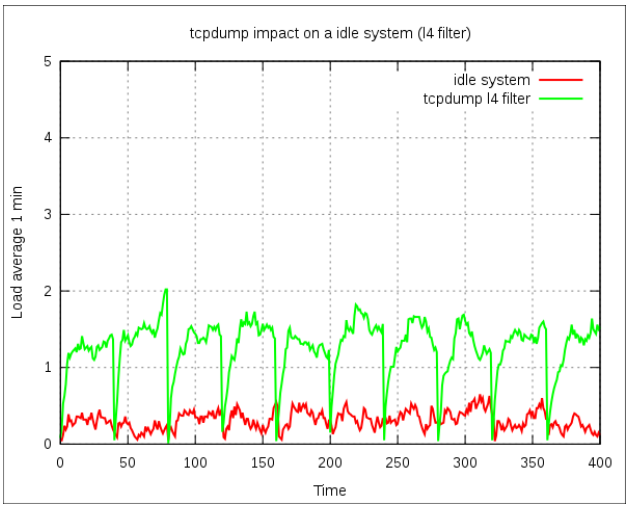


Figura 6.4: *Carga de CPU normal versus carga de CPU com o tcpdump*

	Sistema ocioso	tcpdump (L4)
Tráfego de rede em GBps		
Mínimo	3.3365920	3.7497874
Máximo	9.3894851	9.3882138
Média	9.2610751	8.2228348
Carga média de CPU (1 minuto)		
Mínimo	0.0400	0.0100
Máximo	0.6500	2.0200
Média	0.3108	1.2732

Tabela 6.3: *Valores coletados na comparação do sistema ocioso e o tcpdump com filtro de camada 4*

Como pode-se observar na tabela 6.3, a variação entre este teste e o primeiro é praticamente nula.

6.4 Impacto do netsfs no sistema

O *netsfs*, por se tratar de um sistema de arquivos, opera de forma bem diferente do *tcpdump*. Ele fica coletando pacotes e incrementando seus contadores internos a partir do momento em que é *montado* em seu ponto de montagem.

6.4.1 netsfs apenas montado

A captura de pacotes para análise posterior no *netsfs* se dá abrindo um descritor de arquivos e lendo (consumindo) seu conteúdo. Em espaço de usuário uma aplicação lê este arquivo e à medida que novos pacotes vão chegando no dispositivo de redes, o kernel "produz"neste mesmo arquivo, criando assim um produtor e consumidor.

Por este motivo, foi feita a medição também com o sistema de arquivos *netsfs* apenas montado, mesmo que neste momento não haja a captura de pacotes para análise e sim apenas para fins de estatísticas.

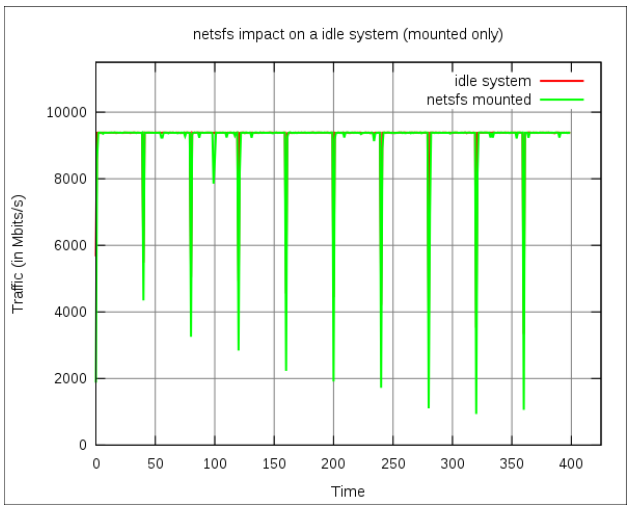


Figura 6.5: *Tráfego de rede normal versus tráfego de rede com o netsfs montado*

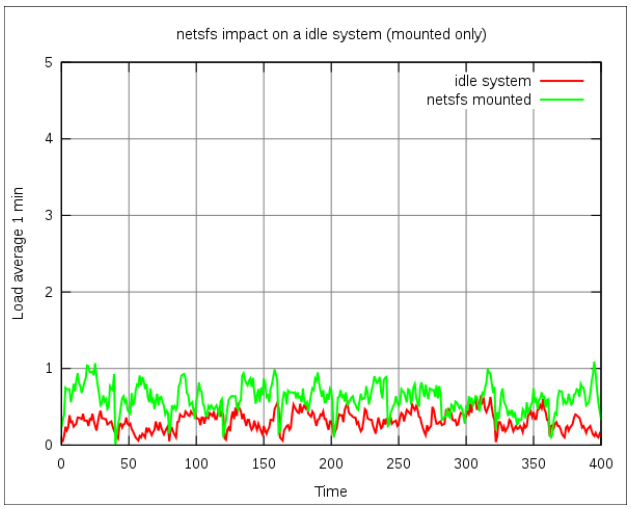


Figura 6.6: *Carga de CPU normal versus carga de CPU com o netsfs montado*

	Sistema ocioso	netsfs montado
Tráfego de rede em GBps		
Mínimo	3.3365920	0.9325
Máximo	9.3894851	9.3902
Média	9.2610751	9.1739
Carga média de CPU (1 minuto)		
Mínimo	0.0400	0.0000
Máximo	0.6500	1.0900
Média	0.3108	0.6110

Tabela 6.4: *Valores coletados na comparação do sistema ocioso e o netsfs montado*

Com os dados da tabela 6.4, e da Figura 6.6 percebe-se que com o *netsfs* apenas montado a vazão da rede fica próximo do ideal (quando o sistema está ocioso). Porém existe um aumento na carga da CPU, pois o mesmo está fazendo a classificação em espaço de kernel para incrementar os contadores.

6.4.2 netsfs com filtro em camada 2

Para que o *netsfs* se comporte de forma parecida com o *tcpdump*, no teste *tcpdump com filtro em camada 2*,basta que o usuário abra o arquivo /net/stream e consuma seu conteúdo.

Isso irá mostrar todos os pacotes que chegam nas interfaces *ethernet*.

Neste experimento, o utilitário *cat* ficou lendo constantemente o conteúdo do arquivo */net/stream*, um pipe provido pelo *netsfs*.

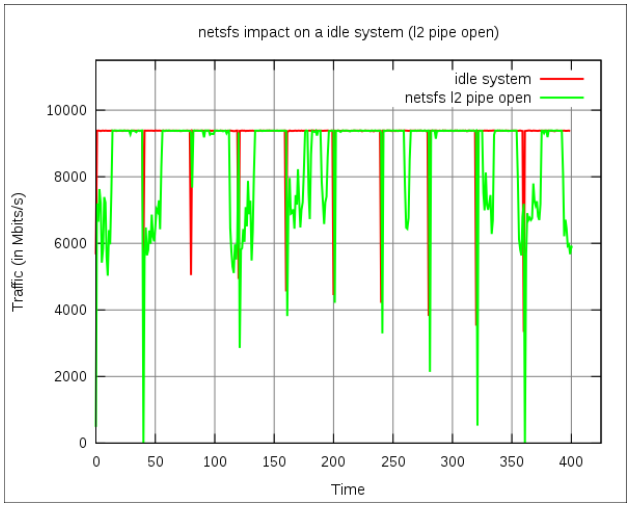


Figura 6.7: *Tráfego de rede normal versus tráfego de rede com o netsfs e com o arquivo /net/stream aberto*

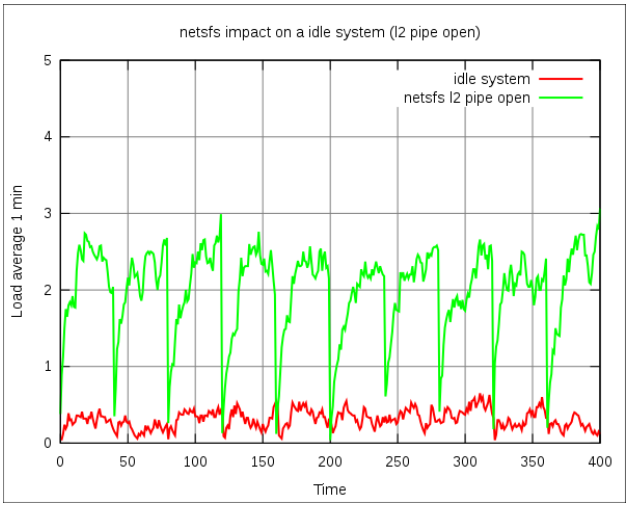


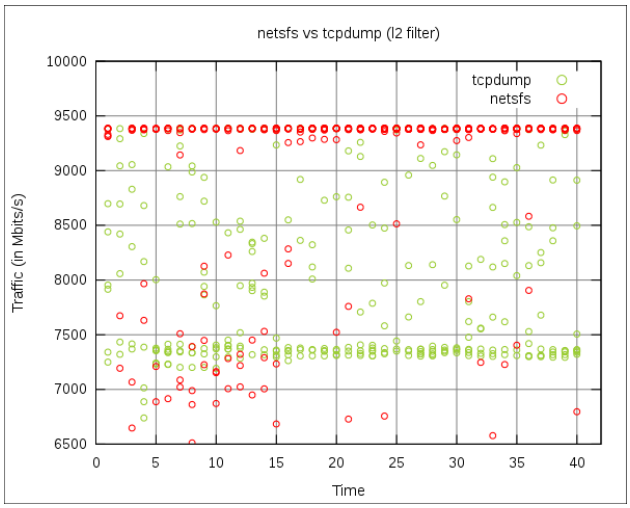
Figura 6.8: *Carga de CPU normal versus carga de CPU com o netsfs e com o arquivo /net/stream aberto*

A tabela 6.5 mostra um resumo dos dados coletados:

	Sistema ocioso	tcpdump (L2)	netsfs (L2)
Tráfego de rede em GBps			
Mínimo	3.3365920	0.5596651	0.4880998
Máximo	9.3894851	9.3882826	9.3902992
Média	9.2610751	8.1576127	8.5206639
Carga média de CPU (1 minuto)			
Mínimo	0.0400	0.1000	0.0400
Máximo	0.6500	1.9300	3.0600
Média	0.3108	1.2118	2.0159

Tabela 6.5: *Valores coletados na comparação do sistema ocioso, tcpdump com filtro de camada 2 e netsfs com filtro de camada 2*

Como a média não representa muito bem os valores coletados, uma distribuição de frequência foi feita para comparar os dois sistemas:



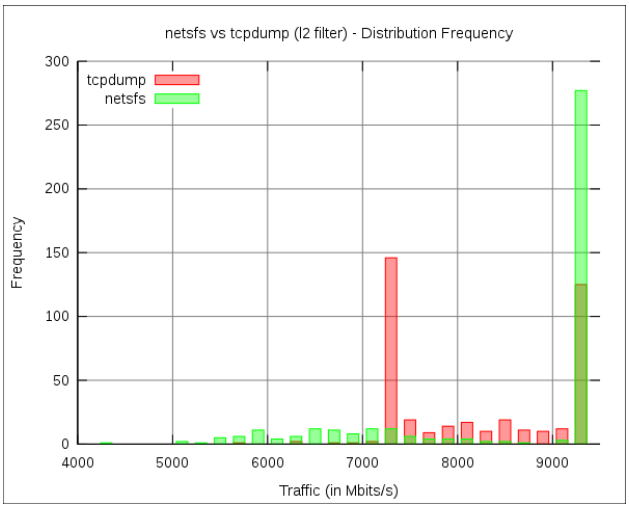


Figura 6.9: *Distribuição de frequência das 400 amostras de tráfego de rede (netsfs vs tcpdump)*

Os dados preliminares da Figura 6.9, mostram que o *netsfs* consegue uma vazão de rede maior

do que o *tcpdump*, atingindo o limite da interface na maioria das amostras.

Por outro lado, em termos de carga de CPU o *netsfs* consome um pouco mais, chegando a 3 processos ficarem aguardando na fila de execução. Isso se deve ao fato do *netsfs* possuir um *lock* em espaço de kernel para garantir algumas transações (como escrita no *FIFO* e cópia das estruturas *skbuff*).

6.4.3 netsfs com filtro em camada 4

Para fazer o *netsfs* capturar pacotes baseando-se na porta de destino basta abrir um arquivo *stream* que encontra-se em um diretório da aplicação (porta). Neste caso o teste foi feito com o arquivo */net/ipv4/tcp/iperf/stream* aberto.

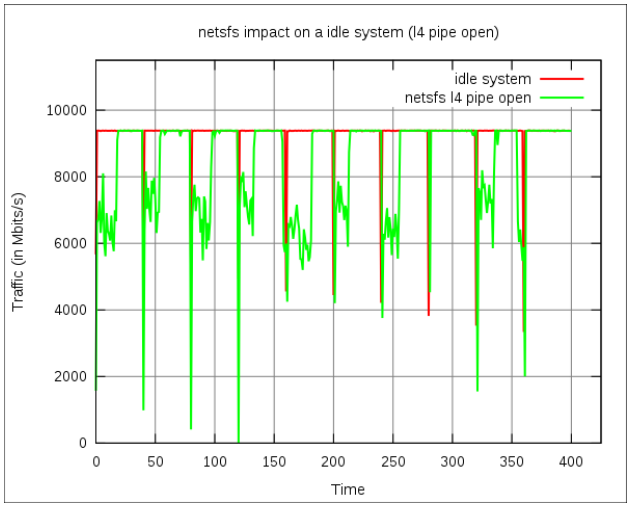


Figura 6.10: *Tráfego de rede normal versus tráfego de rede com o netsfs e com o arquivo /net/ipv4/tcp/iperf/stream aberto*

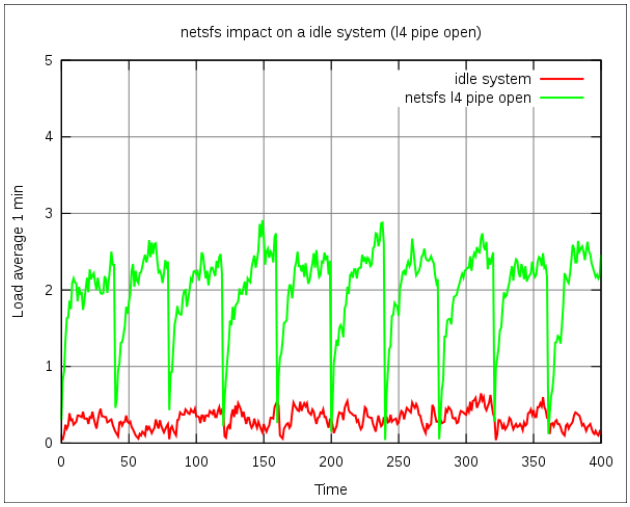


Figura 6.11: *Carga de CPU normal versus carga de CPU com o netsfs e com o arquivo /net/ipv4/tcp/iperf/stream aberto*

A tabela 6.6 mostra um resumo dos dados coletados:

	Sistema ocioso	tcpdump (L4)	netsfs (L4)
Tráfego de rede em GBps			
Mínimo	3.3365920	0.5596651	0.4090172
Máximo	9.3894851	9.3882826	9.3877779
Média	9.2610751	8.1576127	8.3887020
Carga média de CPU (1 minuto)			
Mínimo	0.0400	0.1000	0.0400
Máximo	0.6500	1.9300	2.9100
Média	0.3108	1.2118	2.0251

Tabela 6.6: *Valores coletados na comparação do sistema ocioso, tcpdump com filtro de camada 4 e netsfs com filtro de camada 4*

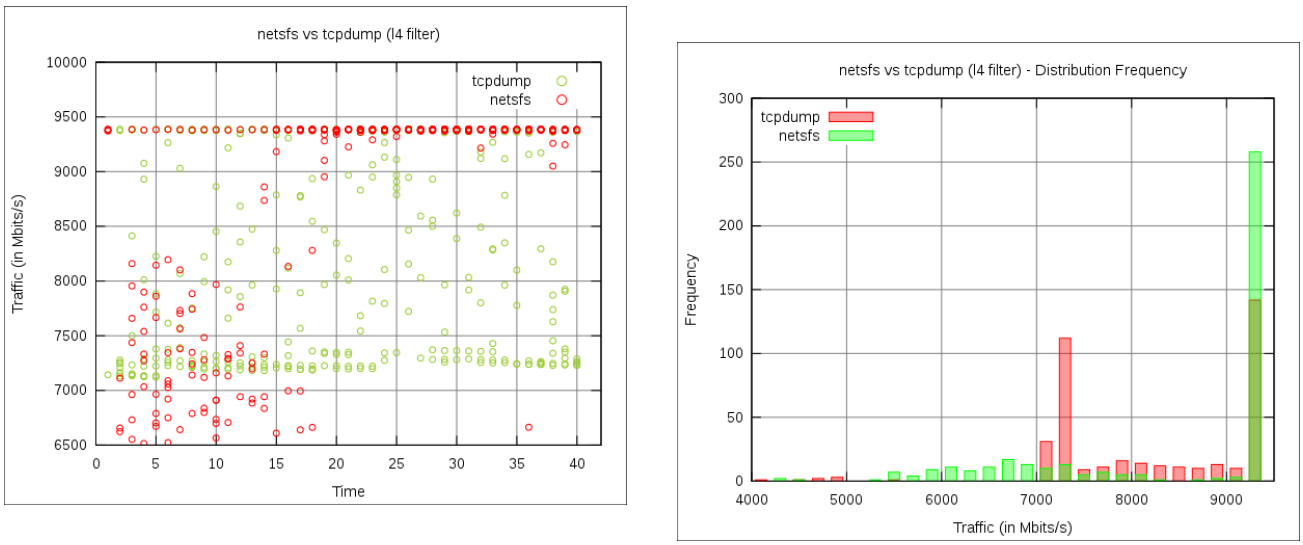


Figura 6.12: *Distribuição de frequência das 400 amostras de tráfego de rede (netsfs vs tcpdump) com filtro em camada 4*

O filtro de camada 4 é mais custoso para o sistema do que o filtro de camada 2. No primeiro, o *netsfs* precisa ler todo o pacote para descobrir quais portas e classificar o pacote quanto ao protocolo da camada 5. Embora ele esteja classificando o protocolo da camada 5 (http, ssh, ftp, etc...) chamamos de filtro de camada 4 pois é preciso ler os dados da camada 4 para obter esta porta.

A Figura 6.12 mostra que o *netsfs* consegue uma vazão de pacotes maior que o tcpdump. Mais de 60% das amostras coletadas ficaram com vazão superior a 9200 Mbps, no caso do *netsfs*. Enquanto que o *tcpdump* mostrou apenas 35% das amostras com esta mesma vazão.

Capítulo 7

Conclusões

Os sistemas operacionais recentes tem usado cada vez mais a interface do sistema de arquivos para a exibição de dados das mais diversas fontes. Apresentamos no capítulo 4 uma arquitetura para exibição e classificação dos pacotes de redes usando esta técnica. Esta arquitetura foi implementada no sistemas de arquivos *netsfs*, exibido no capítulo 5.

Os sistemas operacionais atuais disponibilizam recursos como *zero-copy*, *netlink* e *raw sockets*, que permitem aplicações capturarem de forma fácil os pacotes de rede em espaço de usuário. Embora a implementação do *netsfs* tenha se dado inteiramente em espaço de kernel, uma implementação em espaço de usuário também seria possível, embora existam possíveis perdas de desempenho.

Os resultados apresentados no capítulo 6 mostram que o sistema de arquivos montado e fazendo apenas a contagem dos pacotes possui um impacto mínimo no sistema, ilustrado pelas Figuras 6.5 e 6.6 Embora exista atualmente uma maior atualização da CPU ao capturar pacotes, a vazão da rede apresentou um ganho considerável, mesmo em uma interface de 10 Gbps. Com o aumento no número de núcleos e consequente capacidade das CPUs, este impacto sobre a carga pode ser compensado pela maior vazão, principalmente se considerarmos os casos onde a demanda de rede é maior.

O uso da interface do sistema de arquivos para captura, exibição e filtragem dos pacotes facilitará o tratamento e análise dos dados a partir de aplicações em espaço de usuário.

7.1 Trabalhos futuros

Existem diversas possibilidades de trabalhos futuros tanto no sistema de arquivos de referência proposto, quanto na análise dos dados comparativos. Abaixo estão algumas possibilidades:

- Melhorar o suporte a *IPv6*: Atualmente o *netsfs* suporta *IPv6*, porém não conta com um decodificador da camada de transporte para *IPv6*. Melhorar este suporte é fundamental.
- Adicionar uma classificação baseada em endereço *IP* de origem e destino: Conforme mostrado no capítulo 4, a classificação atual é baseada em protocolos e no caso dos protocolos de camada 5, baseadas nas portas (que por sua vez identificam estes protocolos). Atualmente um filtro baseado em endereços *IP* não está implementado no *netsfs*.
- Melhorar a classificação dos protocolos de camada 5: Atualmente o *netsfs* classifica os protocolos de camada 5 analisando as portas de origem e destino dos pacotes e comparando com uma lista interna de aplicações cadastradas. Permitir uma fácil manutenção desta lista é uma característica desejável ao sistema.
- Implementar suporte à chamada de sistemas *seek()* no arquivo *stream*: Na versão atual do *netsfs* a leitura do buffer se dá por meio das chamadas de sistemas *open()* e *read()*. Ferramentas como *tail* utilizam além destas chamadas a função *lseek()*, que lê o arquivo porém com reposicionamento de ponteiros (*offset*).

- Desenvolvimento de módulos: Conforme descrito no capítulo 4, a arquitetura proposta prevê o suporte à módulos. Porém não faz parte do escopo deste trabalho a exploração destes módulos. Alguns módulos podem ser desenvolvidos para testar este recurso da arquitetura:
  - Sistema de Detecção de Intrusos (IDS) baseados em *HASH match*
  - Captura e armazenamento de arquivos de áudio para tráfego *VoIP*
- Empacotamento para distribuições GNU/Linux: Com o propósito de aumentar o uso do sistema de arquivos aqui proposto, novos usuários e desenvolvedores são fundamentais. Para isso a facilidade da instalação da ferramenta é um fator crucial para sua adoção. O empacotamento do *netsfs*, inicialmente, para distribuições *Debian Like* e *Red Hat Like* precisa ser feito.
- Implementação em espaço de usuário: Ao longo do desenvolvimento deste projeto, foram encontradas técnicas e ferramentas, no qual o desenvolvimento em espaço de usuário é viável. Uma possível implementação do *netsfs* seria em espaço de usuário.
- Comparação entre espaço de *kernel* e usuário: Após o desenvolvimento de uma implementação de referência em espaço de usuário, conforme descrito no item anterior, comparar o desempenho das duas implementações é um caminho natural.

Referências Bibliográficas

[Ben05a] C. Benvenuti. *Understanding Linux Network Internals*. O'Reilly, 2005. 9

[Ben05b] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, primeira edição, 2005. 6

[BHA09] V. BHADRA. Napi - linux new api. http://knol.google.com/k/napi-linux-new-api, Julho 2009. 1, 9, 10

[DLP93] Phil Winterbottom David L. Presotto. The organization of networks in plan 9, 1993. 3

[DM07] A. Dabir e A. Matrawy. Bottleneck analysis of traffic monitoring using wireshark. Em *Innovations in Information Technology, 2007. IIT '07. 4th International Conference on*, páginas 158–162, 2007. 1

[ea90] Rob Pike et al. Plan 9 from bell labs. Em *Proceedings of the Summer 1990 UKUUG Conference*, páginas 1–9. ISBN 0-9513181-7-9, 1990. 3, 6

[eMC05] Daniel P. Bovete e Marco Cesati. *Understanding the Linu Kernel*. O'Reilly Media, third edição, 2005. 7

[Gun10] Dr. Neil J. Gunther. Unix load average part 1 how it works. http://www.teamquest.com/resources/gunther/ldavg1.shtml, Janeiro 2010. 23

[IEE01] IEEE. *IEEE Standard for Information Technology: Portable Operating Syte Interface (POSIX). Part 1: System Interface*. USA: IEEE Standards Association, 2001. 18

[Ins02a] G. Insolvibile. Inside the linux packet filter. *Linux Journal*, 94, Fevereiro 2002. 9

[Ins02b] G. Insolvibile. Inside the linux packet filter, part ii. *Linux Journal*, 95, Março 2002. 9

[Int13] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, vol 1. edição, 2013. 7

[Jai91] Raj Jain. *The art of computer systems performance analysis*. Wiley Professional Computing, primeira edição, 1991. 21

[JE11] Nicolas Bouliane Jan Engelhardt. Writing netfilter modules. http://jengelh.medozas.de/documents/Netfilter_Modules.pdf, 2011. 18

[KH03] Greg Kroah-Hartman. udev - a userspace implementation of devfs. Em *OLS 2003: Ottawa Linux Symposium*, páginas 263–271, Julho 2003. 3

[Kil84] Tom J. Killian. Processes as files. Em *Proceedings of the USENIX Summer Conference*, páginas 203–207. USENIX Association, 1984. 6, 17

[MJ93] Steven McCanne e Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. Em *Proceedings of the USENIX Winter Conference*. USENIX Association, 1993. 1

[MM06] Richard McDougall e Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice Hall, segunda edição, 2006. 6

[Moc05] Patrick Mochel. The sysfs filesystem. Em *OLS 2005: Ottawa Linux Symposium*, páginas 313–326, Julho 2005. 3

[Mou01] Erik J. A. K. Mouw. *Linux Kernel Procfs Guide*, Maio 2001. Delft University of Technology. 6

[mpp] Linux man-pages project. packet(7) - linux man page. http://linux.die.net/man/7/packet. 1

[PJD04] Ravi Prasad, Manish Jain e Constantinos Dovrolis. Effects of interrupt coalescence on network measurements. Em Chadi Barakat e Ian Pratt, editors, *Passive and Active Network Measurement*, volume 3015 of *Lecture Notes in Computer Science*, páginas 247–256. Springer Berlin Heidelberg, 2004. 9

[PN08] Sathish K. Palaniappan e Pramod B. Nagaraja. Efficient data transfer through zero copy. http://www.ibm.com/developerworks/linux/library/j-zerocopy/, Setembro 2008. 1

[Pro81a] DARPA Internet Program. RFC 791: Internet protocol. http://tools.ietf.org/rfc/rfc791.txt, Setembro 1981. 8

[Pro81b] DARPA Internet Program. RFC 793: Transmission control protocol. http://tools.ietf.org/rfc/rfc793.txt, Setembro 1981. 8

[RC05] A. Rubini e J. Corbet. *Linux Device Drivers, 3th edition*. O'Reilly, 2005. 10

[Rea09] L. Correia Villa Real. Uma arquitetura para análise de fluxo de dados estruturados aplicada ao sistema brasileiro de tv digital. Dissertação de Mestrado, Escola Politécnica, Universidade de São Paulo - USP, Brasil, 2009. 3, 4, 13

[Rus00] Paul Rusty Russel. Unreliable guide to hacking the linux kernel. http://www.kernel.org/doc/htmldocs/kernel-hacking.html, 2000. 18

[SGI96] SGI. *IRIX Admin: Disks and Filesystems*, 1996. IRIX 6.2 Reference Manual, Document Number: 007- 2825-001. 6

[SNK+03] J. Salim, Znyx Networks, H. Khosravi, Intel, A. Kleen, Suse, A. Kuznetsov e INR/Swsoft. RFC 3549: Linux netlink as an ip services protocol. http://tools.ietf.org/rfc/rfc3549.txt, Julho 2003. 8

[Sta98] William Stallings. *SNMP,SNMPV2,Snmpv3,and RMON 1 and 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edição, 1998. 22

[STA01] TIA/EIA STANDARD. Tia/eia-568-b.1 commercial building telecommunications cabling standard. http://www.nag.ru/goodies/tia/TIA-EIA-568-B.1.pdf, May 2001. 21

[SWF07] Fabian Schneider, Jöorg Wallerich e Anja Feldmann. Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware. *Springer-Verlag Berlin Heidelberg*, 2007. Deutsche Telekom Laboratories / Technische Universität Berlin Berlin, German. 1

[SZE09] M. SZEREDI. File system in user space - fuse. http://fuse.sourgeforge.net, Março 2009. 17

[Wei84] Paul J. Weinberg. The version 8 network file system. Em *Proceedings of the USENIX Summer Conference*, página 86. USENIX Association, 1984. 6

Índice Remissivo