Cadeia-Aberta: Arquitetura para SFC em Kernel Usando eBPF

Matheus S. Castanho[1], **Cristina K. Dominicini**[2], **Marcos A. M. Vieira**[1]

[1]Universidade Federal de Minas Gerais (UFMG)

{matheus.castanho,mmvieira}@dcc.ufmg.br

[2]Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo (IFES)

cristina.dominicini@ifes.edu.br

Abstract. *RFC 7665 proposes a reference architecture for Service Function Chaining (SFC) that splits all SFC functionality into a set of specialized elements. However, this approach relies on the underlying network infrastructure and requires communication between Service Functions (SF) and SFC elements. In this work, we propose Cadeia-Aberta: an architecture in which SFC elements are implemented transparently inside SF's kernel using Extended Berkeley Packet Filters (eBPF). A proof-of-concept prototype demonstrates that this approach allows direct communication between SFs and reduces communication overhead.*

Resumo. *A RFC 7665 propõe uma arquitetura de referência para encadeamento de funções de rede (Service Function Chaining - SFC) que decompõe as funcionalidades de SFC em um conjunto de elementos especializados. Entretanto, essa abordagem é dependente da infraestrutura de redes e requer comunicação entre funções de serviço (FS) e elementos de SFC. Este trabalho propõe Cadeia-Aberta: uma arquitetura na qual os mecanismos de SFC são implementados de forma transparente no kernel das próprias SFs usando extended Berkeley Packet Filters (eBPF). Um protótipo demonstra que essa proposta permite comunicação direta entre FSs e reduz gargalos na comunicação.*

1. Introdução

Nos últimos anos o paradigma de Virtualização de Funções de Rede (*Network Function Virtualization* - NFV) tem permitido a migração de funções de rede implementadas em hardware para uma abordagem baseada em software. Essa mudança traz benefícios como maior flexibilidade, tempo de desenvolvimento reduzido e a possibilidade de se usar hardwares genéricos, como servidores de propósito geral, reduzindo custos [Mijumbi et al. 2016]. Ao mesmo tempo, interconectar tais funções virtualizadas, também chamadas de funções de serviço (FS), permitindo uma execução sequencial encadeada representa um novo desafio. Diversas organizações e grupos de pesquisa têm proposto soluções para esse problema, chamado de encadeamento de funções de rede (*Service Function Chaining* - SFC) [Quinn and Nadeau 2015].

Uma das principais propostas veio da *Internet Engineering Task Force* (IETF) na forma da RFC 7665 [Halpern and Pignataro 2015]. Ela define um protocolo de encapsulamento para SFC e uma arquitetura composta por três tipos elementos de rede:

classificadores, encaminhadores e *proxies*. Esses elementos atuam conjuntamente para mover os pacotes na sequência correta pelas funções. Todos eles atuam sobre um encapsulamento especial, o *Network Service Header* (NSH), especificado pela RFC 8300 [Quinn et al. 2018]. O cabeçalho NSH carrega informação sobre a cadeia sendo executada e é atualizado no decorrer da execução. Visando desacoplar os planos de serviço e de dados, a arquitetura requer um encapsulamento de transporte externo ao NSH, como VXLAN [Mahalingam et al. 2014], GRE [Farinacci et al. 2000] ou Geneve [Gross et al. 2018], por exemplo, utilizado pelo plano de dados apenas para mover o pacote pela rede.

Porém, o desacoplamento alcançado é apenas parcial. As ações dos encaminhadores são comumente integradas em *switches* virtuais, como OVS [Pfaff et al. 2015] e VPP[1]. Essa abordagem depende de dispositivos especializados e requer que o plano de dados tenha conhecimento do NSH e da infraestrutura de encadeamento. Essa dependência tem o potencial de atrasar a adoção e a implementação do encadeamento de funções em ambientes já existentes, além de restringir a arquitetura a certos dispositivos e plataformas. Apesar de trabalhos recentes abordarem esse problema [Castanho et al. 2018], eles tem suas limitações de escalabilidade e desempenho, pois dependem de elementos intermediários para prover o encadeamento. Portanto o desacoplamento entre os planos de serviço e de dados em ambientes SFC ainda é um problema em aberto.

Neste trabalho, é proposto *Cadeia-Aberta*, uma nova arquitetura para habilitar o desacoplamento total entre esses dois planos. Nela, os componentes de SFC são integrados dentro de cada função, sendo efetivamente transformados em estágios de processamento. Dessa forma, as ações necessárias para habilitar o encadeamento ficam restritas às funções, geralmente implementadas como máquinas virtuais ou contêineres. Assim, o plano de dados não precisa ter conhecimento nenhum sobre a arquitetura de SFC. A viabilidade da arquitetura é provada por meio da construção de um protótipo utilizando a tecnologia *extended Berkeley Packet Filters (eBPF)*. Os estágios são implementados como programas eBPF [Miano et al. 2018] dentro do kernel do Linux, de forma que as operações de SFC são feitas durante o processamento dos pacotes pela pilha de rede do sistema operacional (SO). Essa abordagem faz com que a arquitetura seja completamente transparente não só à rede, mas também às funções de rede virtualizadas, executadas em espaço de usuário.

O restante deste documento está organizado da seguinte forma: a seção 2 provê uma contextualização de SFC e eBPF. A seção 3 discute trabalhos relacionados. O projeto e implementação da arquitetura proposta são discutidos na seção 4. A seção 5 descreve a avaliação experimental. A seção 6 discute as vantagens e desvantagens da arquitetura proposta e a seção 7 conclui esse trabalho.

2. Contextualização

Nesta seção, são apresentadas a arquitetura de referência para SFC proposta pela IETF (seção 2.1) e a tecnologia eBPF (seção 2.2), utilizada no protótipo apresentado.

[1]https://fd.io/technology/

2.1. Arquitetura da RFC 7665

Recentemente o grupo de trabalho de SFC da IETF vem propondo diversos padrões para SFC, dos quais pode-se destacar a RFC 7665 [Halpern and Pignataro 2015], que especifica uma arquitetura de referência para SFC, e a RFC 8300 [Quinn et al. 2018] que padroniza um protocolo de encapsulamento a ser usado nessa arquitetura, o *Network Service Header* (NSH). Um cabeçalho NSH carrega dois identificadores: o *Service Path Identifier* (SPI) de 3 bytes e o *Service Index* (SI) de 1 byte. O primeiro indica a cadeia em execução, representada por um número identificador. O segundo informa qual é o próximo elemento da cadeia a ser executado, representado por um índice de 0 a 255. O SI é decrementado após a execução de cada função de serviço, e é utilizado conjuntamente com o SPI na escolha do próximo salto da cadeia pelos encaminhadores.

Na arquitetura da RFC 7665, representada na Figura 1, o encadeamento das funções é habilitado a partir da operação conjunta de 4 tipos de elementos:

- **Classificadores**: pacotes entrando no ambiente SFC são classificados para determinar qual encadeamento deve ser executado, seguido da inserção do cabeçalho NSH correspondente;
- **Funções de serviço (FS)**: executa algum processamento sobre o pacote. Podem ser divididas em duas categorias: *NSH-aware* e *NSH-unaware*. A primeira é composta por funções que têm conhecimento do encapsulamento NSH, e sabem operar sobre ele. As funções da segunda, também chamada de *legadas*, não são cientes do protocolo, e necessitam da intermediação de um *proxy*;
- ***Proxies***: são responsáveis pela remoção e reinserção do cabeçalho NSH nos pacotes antes e depois, respectivamente, do seu processamento por funções *NSH-unaware*;
- **Encaminhadores**: a cada etapa da execução de uma cadeia, o encaminhador é responsável por decidir qual é a próxima função a ser executada e modificar o encapsulamento de transporte apropriadamente para permitir que a rede consiga entregar o pacote ao próximo salto.

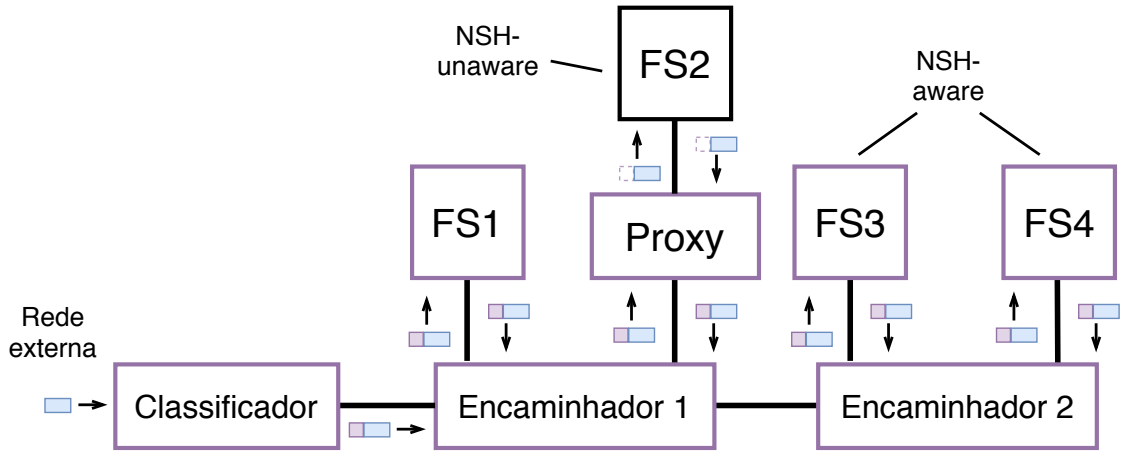


Figura 1. Exemplo de arquitetura segundo a RFC 7665

2.2. Extended Berkeley Packet Filter (eBPF)

BPF é um conjunto de instruções inicialmente criado para auxiliar na captura e filtragem de pacotes no sistema BSD [McCanne and Jacobson 1993]. Criada por Steven McCanne e Van Jacobson, é componente fundamental da biblioteca *libpcap*, amplamente utilizada por aplicações de rede, como *tcpdump* e Wireshark [Orebaugh et al. 2006]. Programas escritos com essa linguagem são executados por máquinas virtuais implementadas dentro do kernel. Com isso, é possível alterar o processamento de pacotes feito pela pilha de protocolos do sistema operacional, adicionando um nível de programabilidade ao processamento de pacotes.

Após a sua criação, a linguagem BPF foi portada para o sistema operacional Linux, onde mais recentemente, em 2013, recebeu uma grande atualização com novas funcionalidades[2]. Essa nova versão foi proposta por Alexei Starovoitov, e é denominada *extended Berkeley Packet Filter* (eBPF). Com o tempo, a nova versão da linguagem naturalmente herdou o nome BPF, enquanto a anterior hoje é chamada de *classic BPF* (cBPF).

As diferenças entre cBPF e eBPF são grandes, desde o número de registradores disponíveis (de 2 para 11), possibilidade de encadeamento de programas BPF utilizando *tail calls*, além de todo um arcabouço de funções auxiliares e ferramentas providas pelo sistema Linux. Porém, uma das mais importantes é a adição de estruturas chave-valor, denominadas mapas, algo não disponível na versão original. Atualmente, o kernel do Linux apresenta diversos mapas distintos, como tabelas *hash*, *arrays*, pilhas, filas, tabelas de redirecionamento, entre outras. Além de aumentar a aplicabilidade de programas BPF, esses mapas podem ser acessados tanto de espaço de kernel quanto por aplicações em espaço de usuário, permitindo troca de informação entre os dois meios.

3. Trabalhos relacionados

Alguns trabalhos recentes abordam o desacoplamento entre os planos de serviço e de dados. *PhantomSFC* [Castanho et al. 2018] transforma todos os componentes de SFC em funções individuais, ao invés de papéis lógicos desempenhados por outros dispositivos de rede. Essas funções especiais executam ao lado das demais funções, porém atuam apenas nas ações necessárias para habilitar o encadeamento. Como todo pacote precisa passar pelo encaminhador a cada passo de execução de uma cadeia, o resultado da virtualização dos elementos de SFC é a criação de uma topologia do tipo estrela centrada nos encaminhadores. Portanto, esses elementos se tornam potenciais gargalos, além de importantes pontos de falha. O mesmo problema se aplica aos *proxies*, em menor grau.

Outra técnica proposta para habilitar SFC que tem sido discutida atualmente é o uso de roteamento por segmento (*segment routing*) [Abdelsalam et al. 2017, Duchene et al. 2018] utilizando o protocolo IPv6. Nessa abordagem um pacote é encapsulado com várias *labels*, na forma de uma pilha, que definem como o pacote deve ser encaminhado na rede, especificando assim o encadeamento de funções. A cada salto, os dispositivos de rede leem a próxima *label* e encaminham o pacote correspondentemente. Essa abordagem diminui a complexidade da arquitetura de encadeamento, já que não existe a necessidade de elementos encaminhadores e configuração de regras de encadeamento na rede, apenas nas pontas. Porém essa abordagem é ainda mais acoplada

[2]https://lwn.net/Articles/740157/

aos dispositivos de rede do que a proposta pela RFC 7665, já que os dispositivos intermediários necessitam operar sobre as *labels*.

Diversos outros trabalhos usando a tecnologia eBPF têm surgido, muitos inspirados pelo projeto de código aberto IOVisor[3]. InKev [Ahmed et al. 2018] habilita a execução de programas no plano de dados de redes virtuais, com foco em redes de centros de dados. Tu *et al.* [Tu et al. 2017] descrevem o projeto, implementação e avaliação de um plano de dados extensível para OvS, baseado em eBPF. Para permitir que o protocolo *OpenFlow* suporte protocolos diversos, Jouet *et al.* [Jouet et al. 2015] definem uma extensão para o protocolo, chamada *OpenFlow Extended Match field* (OXM), para instalar *bytecode* BPF. Os autores também adicionaram o mecanismo da biblioteca *libpcap* a um switch OpenFlow para executar esse *bytecode* [Jouet et al. 2015]. Xhonneaux *et al.* [Xhonneux et al. 2018] também utiliza programas eBPF para habilitar a arquitetura de SFC utilizando roteamento por segmento, discutido anteriormente. No contexto de segurança, Betrone *et al.* [Bertrone et al. 2018] propõem uma nova ferramenta *iptables* baseada em eBPF. Nesse trabalho é apresentada uma nova aplicação dessa tecnologia, dessa vez para separar os planos de dados e serviço em ambientes de SFC.

4. Cadeia-Aberta

Para desacoplar a arquitetura de encademento da rede, é necessário retirar as ações de SFC do plano de dados, de forma que os dispositivos de rede não precisem ter nenhum conhecimento sobre o mecanismo de encadeamento. Nesta seção é apresentado uma nova solução para esse problema, a arquitetura *Cadeia-Aberta* (4.1). Em seguida, é discutido a implementação de um protótipo utilizando eBPF (4.2).

4.1. Arquitetura

Visando habilitar o desacoplamento total, pode-se retirar as ações dos elementos de SFC dos dispositivos do plano de dados e integrá-las às máquinas virtuais que executam as funções de rede. Essas ações podem ser resumidas nas seguintes operações: classificação de pacotes, encapsulamento, desencapsulamento e encaminhamento.

Idealmente, essa integração deve ser transparente à função, evitando a necessidade de alterações na implementação ou funcionalidade da mesma. Isso pode ser feito na forma de estágios de pré e pós-processamento adicionados a cada função de rede virtualizada, como ilustrado na Figura 2. Operações de encaminhamento, tradicionalmente efetuadas por um encaminhador, podem ser feitas localmente por um estágio de encaminhamento adicionado ao fim da execução do pacote por cada função. Dessa forma, o elemento de encaminhamento se torna distribuído, e cada função é responsável por ajustar o cabeçalho externo do pacote para fazer o encaminhamento para a próxima função na cadeia. Com isso, a arquitetura deixa de ser centrada nos encaminhadores e passa a ter uma topologia mais genérica, sem a necessidade de constantes trocas de mensagens entre funções e encaminhadores.

Similarmente, as operações de desencapsulamento e encapsulamento feitas pelos *proxies* podem ser implementadas por estágios adicionados exatamente antes e depois,

[3]www.iovisor.org

respectivamente, da execução da função em si. A Figura 2 ilustra uma função de rede com os múltiplos estágios locais.

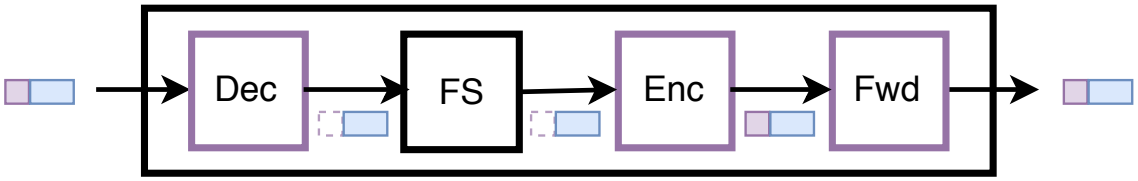


Figura 2. Exemplo de função de rede com os estágios extras de processamento SFC

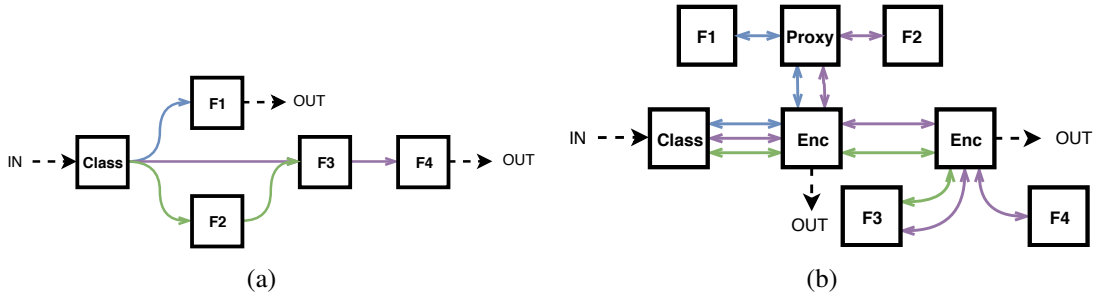
Como mostrado na Figura 2, uma função nessa nova arquitetura é composta por quatro estágios de processamento:

1. **Desencapsulamento (*Dec*)**: ao receber um pacote encapsulado com NSH, este estágio é responsável por fazer o desencapsulamento, retirando o NSH e salvando essa informação para posterior readição no estágio de encapsulamento;
2. **Função (*FS*)**: esse é o estágio responsável pela real funcionalidade da função de serviço. Nele é implementada a lógica de tratamento de pacotes principal da função. No caso de uma função de NAT, por exemplo, nesse estágio seriam efetuadas as mudanças de endereços IP e portas;
3. **Encapsulamento (*Enc*)**: após a execução da função, o cabeçalho NSH precisa ser readicionado ao pacote. Esse estágio é responsável por reinserir o cabeçalho retirado anteriormente e decrementar o campo SI;
4. **Encaminhamento (*Fwd*)**: nesse estágio são implementas as ações do encaminhador. Os índices SPI e SI do cabeçalho NSH são utilizados para consultar uma tabela de encaminhamento e decidir qual é o endereço da próxima função da cadeia. Após a consulta, o encapsulamento de transporte é atualizado com esse novo endereço e o pacote é entregue à rede.

O exemplo dado considera que a função de rede sendo executada pelo estágio *FS* é *NSH-unaware*, isto é, não tem conhecimento sobre o protocolo NSH. Na arquitetura da RFC 7665, esse tipo de função necessita de um *proxy* para ser integrado ao ambiente de SFC. Para funções *NSH-aware*, nas quais não é necessária a retirada do NSH, os estágio *Dec* e *Enc* podem ser desabilitados. Nessa situação, pacotes recebidos seriam diretamente entregues ao estágio *FS* e em seguida ao estágio *Fwd*, com posterior reenvio à rede.

Com a utilização dos estágios, não há mais necessidade de encaminhadores intermediários entre as funções. Essas podem se organizadas em qualquer tipo de topologia lógica entre as FSs, que é definida pelas regras de encaminhamento instaladas diretamente no estágio *Fwd*. Essa diferença na topologia pode ser observada na Figura 3.

O cenário ilustrado é composto por quatro funções. Na arquitetura *Cadeia-Aberta*, cada função é como uma caixa fechada, com todas as funcionalidades necessárias, e a comunicação entre as funções que compõem uma cadeia é feita de forma direta. Em contraste, na arquitetura de referência da RFC 7665, mostrada na Figura 3(b), a comunicação entre as funções é feita de forma indireta, sempre passando por um encaminhador e possivelmente por um *proxy*, como é o caso das funções *F1* e *F2*. Todas as regras necessárias para configurar os estágios existentes e habilitar o encadeamento de funções são adicionadas por um plano de controle, que está fora do escopo deste trabalho.

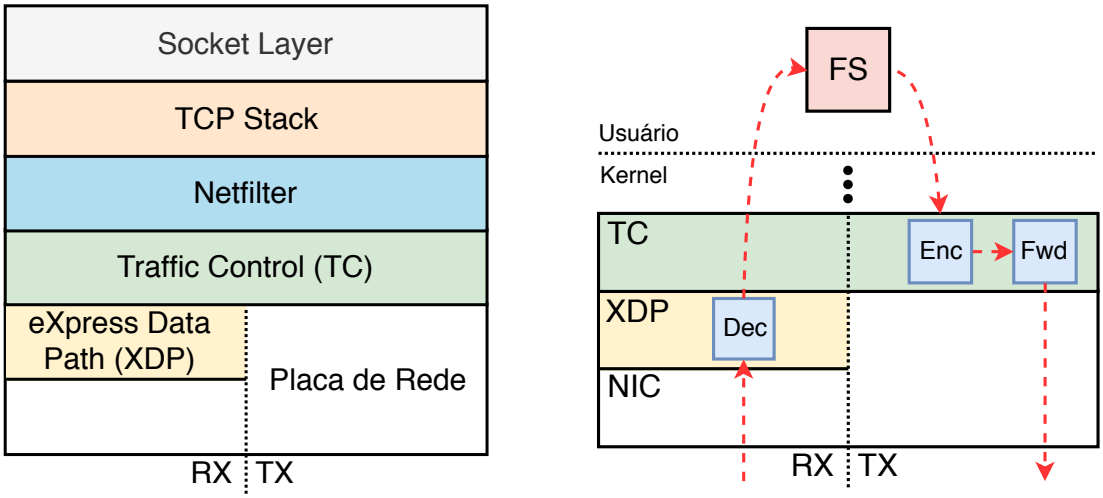


(a) (b)

Figura 3. Exemplos de cenários utilizando duas arquiteturas (a) Arquitetura Cadeia-Aberta (b) Arquitetura de referência da RFC 7665

4.2. Implementação

Para implementar os estágios de processamento, foi criado um protótipo utilizando programas eBPF carregados no kernel do Linux. Essa tecnologia foi escolhida por possibilitar a alteração do processamento de pacotes em espaço de kernel, o que efetivamente torna os estágios transparentes às funções. Dessa forma, não é necessária nenhuma alteração na implementação da função, como desejado.



(a) (b)

Figura 4. (a) Camadas da pilha de rede do kernel do Linux (b) Estágios implementados como códigos BPF no kernel do Linux

Códigos BPF podem ser carregados em diversos ganchos (*hook points*) no kernel, que é composto por diversas camada de processamento de pacotes, como mostra a Figura 4(a). Programas BPF associados a cada camada veem diferentes contextos (isto é, o pacote a que tem acesso) e têm permissão de uso de diferentes conjuntos de funções auxiliares. Por exemplo, programas na camada *eXpress Datapath* (XDP) são os primeiros a interagirem com os pacotes após o recebimento pela placa de rede. O contexto para eles é um quadro de camada 2, ou seja, o pacote inteiro. Além disso, estão disponíveis funções de modificação do tamanho do pacote, inserção e retirada de cabeçalhos, reescrita de campos, entre outras. Já na camada Netfilter, o contexto recebido pelo programa é um pacote de camada 3, sendo inacessível o cabeçalho Ethernet. Em contrapartida, é

permitido o uso de novas funções auxiliares para cálculo de checksum e tunelamento, por exemplo, não disponíveis na camada XDP.

Por ser a mais próxima à placa de rede, a camada XDP se torna o gancho ideal para a implementação dos estágios. Durante a recepção no XDP, o sistema operacional ainda não criou *socket buffers* nem metadados para o pacote, sendo possível fazer alterações arbitrárias no mesmo antes de entregá-lo para as camadas superiores. Ela apresenta o maior desempenho se comparada às outras [Miano et al. 2018].

Porém, o propósito dessa camada é prover uma forma rápida de tratar pacotes recebidos, possibilitando, por exemplo, o descarte antecipado de pacotes, antes que esses sejam entregues à pilha. Por conta disso, programas eBPF associados à camada XDP só são capazes de processar pacotes na recepção, e não na transmissão. Portanto, o único estágio capaz de ser implementado nesse nível é o de desencapsulamento (*Dec*).

Logo acima do XDP, está a camada de *Traffic Control* (TC), que é responsável pela implementação e aplicação de políticas de controle de tráfego no Linux. Por meio dela é possível ajustar as políticas de fila utilizadas pelo sistema operacional, assim como aplicar filtros e ações personalizadas para o tratamento de pacotes. Nessa camada, programas eBPF representam um tipo especial de filtro, que é programável e capaz de determinar que ação deve ser executada sobre o pacote (ex: aceitar, rejeitar, abortar, adicionar túnel). O contexto visto por tais programas também é o de pacotes de camada 2, portanto têm acesso a todos os cabeçalhos.

Na transmissão, essa é a camada mais próxima da placa de rede, representando um dos últimos trechos de código a serem executados sobre o pacote antes do envio ao *driver* da interface de rede. Os dois estágios restantes, o de encapsulamento (*Enc*) e o de encaminhamento (*Fwd*) foram implementados nessa camada.

Diferentemente da camada XDP, programas eBPF no TC não tem acesso a funções de encapsulamento/desencapsulamento que possibilitem o uso de protocolos genéricos. A única forma encontrada pelos autores para se adicionar novos bytes ao cabeçalho do pacote foi utilizando diretivas de inserção de rótulos VLAN. Cada chamada à função auxiliar $bpf\_skb\_vlan\_push()$ adiciona 4 bytes ao cabeçalho Ethernet, correspondendo a um novo cabeçalho VLAN. Como o programa tem acesso total aos bytes do pacote, é possível sobrescrever os rótulos VLAN adicionados e reorganizar os cabeçalhos da forma desejada. Este artifício foi utilizado para habilitar o encapsulamento no estágio *Enc*.

Por fim, o estágio *FS*, que implementa a função de serviço em si, é implementado como uma aplicação de rede em espaço de usuário, permitindo assim o uso da maioria das funções já existentes hoje. Uma limitação, porém, é que a implementação atual suporta apenas funções que usam *raw sockets*, pois nesse modelo a aplicação recebe os pacotes logo acima da camada TC. Para funções que utilizam sockets TCP, por exemplo, a captura dos pacotes acontece mais acima da pilha, e talvez seja necessária a implementação de outros programas eBPF em camadas acima do XDP e TC para garantir o tratamento correto dos pacotes.

Diversos pesquisadores sugerem que funções devem ser implementadas em espaço de kernel, visando evitar perda de desempenho com trocas de contexto [Miano et al. 2018, Ahmed et al. 2018]. Porém, por mais polivalentes que programas eBPF no kernel sejam, eles também têm suas limitações como inexistência de *loops* e

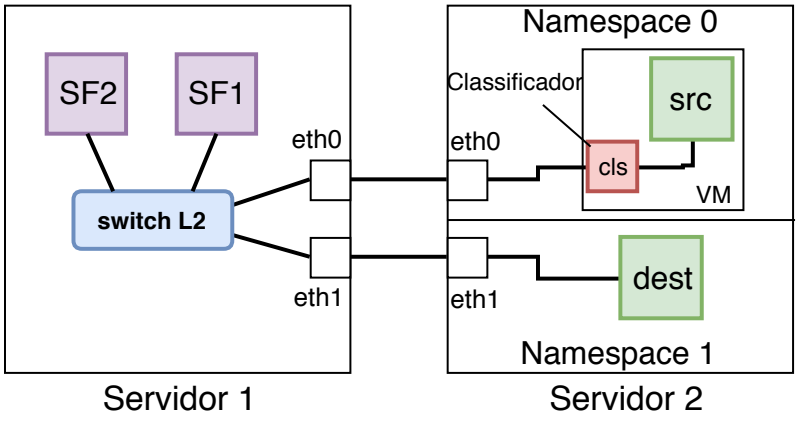


Figura 5. Infraestrutura do ambiente utilizado durante os testes

tamanho reduzido dos programas, de forma que a implementação de funções em espaço de usuário permite uma gama de funcionalidades muito maior. Uma possibilidade para se garantir um equilíbrio entre essas duas abordagens é a utilização de bibliotecas como *Data Plane Development Kit* (DPDK)[4] em conjunto com *sockets* do tipo AF_XDP[5], que permitem o processamento de pacotes em espaço de usuário com alto desempenho e ao mesmo tempo a integração de programas BPF na camada XDP para o tratamento de pacotes.

5. Avaliação experimental

Para avaliar a aplicabilidade da arquitetura proposta, foi criado um ambiente de testes conforme a Figura 5. Foram utilizados dois servidores Dell PowerEdge T430, com processador Intel Xeon E5-2620 v3 @ 2.40GHz e uma placa de rede Intel I350 (quad-port, 1 GbE) cada. O Servidor 1 continha 64 GB memória e o Servidor 2 16 GB.

Neste ambiente a comunicação entre dois hospedeiros, um de origem *src* e outro de destino *dest*, passa por uma cadeia de funções antes de chegarem ao destino. Apenas os pacotes de *src* para *dest* foram encaminhados por dentro da cadeia. O nó *src* foi colocado dentro de uma máquina virtual, na qual foi configurado um elemento Classificador de SFC, que encapsulava o tráfego de saída e adicionava o cabeçalho NSH correspondente para permitir o encadeamento. Esse elemento também foi implementado como um código eBPF associado ao caminho *egress* da camada TC do Linux, sendo completamente transparente para a aplicação cliente.

Todas as funções utilizadas consistiram em um simples fio virtual, que recebia pacotes e os reenviava pela mesma interface, implementado utilizando *switch* OVS com apenas 1 regra. Essa função simples permite eliminar possíveis atrasos adicionados pelo processamento adicional imposto por funções complexas, de forma que os resultados obtidos refletem o desempenho máximo da arquitetura. A plataforma de virtualização utilizada foi QEMU KVM e o switch L2 usado na comunicação foi criado usando Linux *bridges*.

Durante os testes foi analisado o impacto do tamanho da cadeia na latência ponta-

[4]dpdk.org
[5]https://lwn.net/Articles/750293/

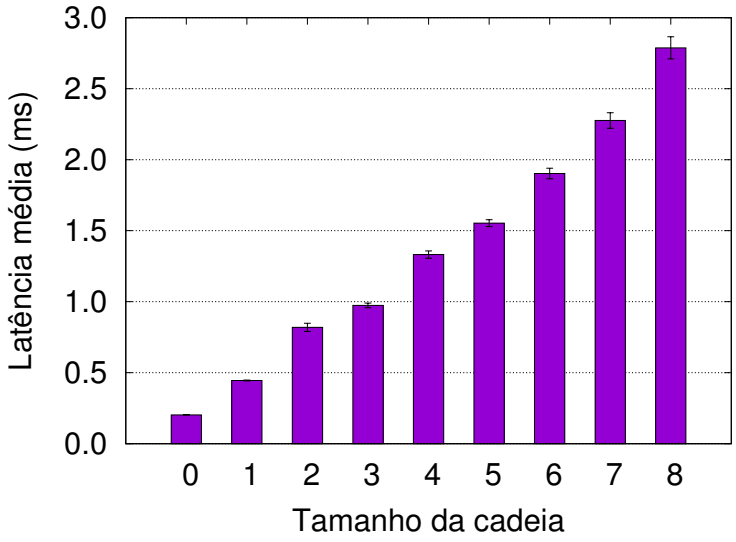


Figura 6. Latência média para diferentes tamanhos de cadeia

a-ponta observada pelo nó de origem utilizando a ferramenta *ping* com ICMP sobre IP de 64 Bytes a uma taxa de 100 pps. Foram avaliados diversos tamanhos de cadeia, variando desde 0 (comunicação sem SFC) até cadeias com 8 funções em sequência. A Figura 6 mostra que há um aumento linear na latência de cerca de 0,2 ms a cada função adicionada. Os resultados representam a média de 15 rodadas, cada uma consistindo de 100 pacotes ICMP. As barras no gráfico representam o erro padrão.

O incremento linear observado só é possível pois as funções se comunicam diretamente. Arquiteturas similares [Castanho et al. 2018] requerem elementos intermediários na comunicação, de forma que cada nova função adicionada à cadeia impõe uma maior carga no encaminhador, tornando o aumento da latência exponencial.

6. Discussão

Além de desacoplar os planos de serviço e de dados, a arquitetura *Cadeia-Aberta* e sua implementação utilizando a tecnologia eBPF apresentam diversas características interessantes. A seguir, são discutidas as principais, enfatizando as principais vantagens e desvantagens da abordagem e da implementação propostas.

Transparência à função e à rede. Ao mover toda a funcionalidade de SFC para o kernel da máquina virtual de cada função, retira-se a necessidade de aplicações (funções de rede) e da infraestrutura terem conhecimento e operarem sobre o encapsulamento NSH. Caso as funções utilizadas sejam *NSH-aware*, a transparência não é uma necessidade, de forma que podemos desabilitar os estágios *Dec* e *Enc*. Porém, funções legadas isto é, *NSH-unaware*, podem fazer parte do encadeamento sem necessitarem de nenhuma alteração no seu código ou modo de operação. Isso é possível pois todas as operações necessárias sobre o encapsulamento NSH são feitas pelos outros estágios de processamento, dentro do kernel.

Maior facilidade de execução de operações de *proxy*. Na arquitetura proposta pela RFC 7665, efetuar operações de *proxy* pode não ser algo trivial. Tais elementos

precisam fazer um casamento entre os pacotes de entrada e saída de uma função de rede. Porém, a arquitetura não impõe nenhuma restrição quanto ao tipo de alterações que as funções podem fazer aos pacotes. Tais modificações podem ir desde modificação de cabeçalhos e carga de dados, até deleção e criação de múltiplos novos pacotes. Essa dinamicidade torna a tarefa de casamento de pacotes de difícil execução, requerendo algoritmos complexos [Qazi et al. 2013], que apesar de eficazes não são totalmente precisos. Ao movermos essas ações para dentro do kernel, tanto as operações quanto a execução da função ocorrem sob o mesmo domínio de software, isto é, o mesmo sistema operacional.

Apesar de fora do escopo deste trabalho, o casamento dos pacotes pode ser facilitado por meio do uso de estruturas presentes no próprio sistema operacional. Por exemplo, se um esquema sem cópia é usado na passagem dos pacotes da placa de rede para o espaço de usuário, qualquer operação sobre o pacote operará sobre o mesmo espaço de memória, tornando o casamento no estágio de encapsulamento uma operação trivial. Além disso, metadados adicionais podem ser adicionados à estrutura do pacote para auxiliar na correspondência entre pacotes.

Maior resiliência a falhas. Como a arquitetura *Cadeia-Aberta* propõe uma maior distribuição das operações de SFC, há uma redução dos pontos de falha do sistema como um todo. Na arquitetura original, a falha em um encaminhador afetaria todas as funções de rede associadas a ele, mesmo que estivessem em perfeito funcionamento. Na abordagem proposta, a falha de qualquer elemento não tem o poder de afetar o funcionamento de outras funções, mas apenas a si mesmo e as cadeias as quais ele pertence. Além disso, apesar de visar os mesmos objetivos da arquitetura PhantomSFC [Castanho et al. 2018], *Cadeia-Aberta* não se limita a topologias do tipo estrela. Na verdade, as cadeias podem formar qualquer tipo de topologia, sem a necessidade de elementos centralizadores intermediários.

Estágios programáveis. Por natureza, máquinas BPF podem ter sua funcionalidade modificada em tempo de execução a qualquer momento. Além disso, a sua linguagem é genérica o suficiente para permitir a implementação de diferentes programas. Isso torna os estágios de processamento efetivamente programáveis. Por exemplo, diferentes técnicas de casamento de pacotes podem ser implementadas e carregadas a qualquer momento nos estágios de *proxy*. O mesmo pode ser feito com o estágio *Fwd*, permitindo a implementação de novas funcionalidades e diferentes formas de encaminhamento sob demanda.

***Offload* para dispositivos programáveis**: atualmente já existem dispositivos capazes de executar códigos eBPF em hardware [Kicinski and Viljoen 2016, Pacífico et al. 2018], assim como métodos nativos ao Linux para fazer *offloading* para tais dispositivos. Isso permite ainda maiores ganhos de desempenho em relação à execução no kernel.

Apesar dos diversos pontos positivos, a arquitetura também apresenta algumas desvantagens, como **necessidade de comunicação do controlador com todas as funções de rede para configuração das regras de encaminhamento**. Ao invés de estabelecer comunicação apenas com alguns encaminhadores e *proxies* para configurar o ambiente, o plano de controle precisará se comunicar com todas as funções habilitadas. Esta exigência, porém, é amenizada pelo fato do plano de controle já precisar configurar as

funções de rede como, por exemplo, para instalar regras em *Firewalls* e *NATs*.

Restrito ao Linux. As mudanças mais recentes no arcabouço eBPF, até o momento, foram implementadas apenas no Linux. Logo, a implementação da arquitetura *Cadeia-Aberta* feita hoje é restrita apenas a esse ambiente. Porém, a comunidade FreeBSD já vem discutindo sobre uma adaptação do eBPF para o sistema[6]. Mesmo assim, essa limitação não causa uma grande restrição à aplicação da arquitetura nos diversos ambientes de nuvem e centro de dados de hoje, visto que a grande maioria desses ambientes rodam Linux. A única restrição seria a utilização de uma versão mais recente do kernel.

7. Conclusão

Neste trabalho foi proposta uma nova arquitetura para SFC, denominada *Cadeia-Aberta*. Visando desacoplar os planos de dados e de serviço, os elementos de SFC da RFC 7665 são transformados em estágios de processamento e integrados a cada função de serviço.

Dessa forma, não há mais a necessidade de elementos intermediários na comunicação entre as funções da cadeia, reduzindo potenciais gargalos e generalizando os tipos de topologia lógica possíveis. Também foi apresentado um protótipo que implementa os estágios como programas eBPF carregados ao kernel do Linux, tornando o mecanismo de SFC completamente transparente às funções e à rede.

Os testes feitos mostram que essa abordagem permite um incremento linear na latência para cada função adicionada a uma cadeia, o que representa uma evolução em relação ao estado da arte [Castanho et al. 2018]. Como trabalhos futuros, espera-se modificar o mecanismo de encapsulamento utilizado para permitir maior desempenho e implementar um plano de controle para interagir com o ambiente.

8. Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001

Referências

[Abdelsalam et al. 2017] Abdelsalam, A., Clad, F., Filsfils, C., Salsano, S., Siracusano, G., and Veltri, L. (2017). Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure. In *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017*, pages 1–5. IEEE.

[Ahmed et al. 2018] Ahmed, Z., Alizai, M. H., and Syed, A. A. (2018). Inkev: In-kernel distributed network virtualization for dcn. *SIGCOMM Comput. Commun. Rev.*, 46(3):4:1–4:6.

[Bertrone et al. 2018] Bertrone, M., Miano, S., Risso, F., and Tumolo, M. (2018). Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, SIGCOMM '18, pages 108–110, New York, NY, USA. ACM.

[6]https://www.bsdcan.org/2018/schedule/track/Hacking/963.en.html

[Castanho et al. 2018] Castanho, M. S., Dominicini, C. K., Villaça, R. S., Martinello, M., and Ribeiro, M. R. N. (2018). Phantomsfc: A fully virtualized and agnostic service function chaining architecture. In *Computers and Communications (ISCC), 2018 IEEE Symposium on*. IEEE.

[Duchene et al. 2018] Duchene, F., Jadin, M., and Bonaventure, O. (2018). Exploring various use cases for ipv6 segment routing. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, SIGCOMM '18, pages 129–131, New York, NY, USA. ACM.

[Farinacci et al. 2000] Farinacci, D., Li, T., Hanks, S., Meyer, D., and Traina, P. (2000). Generic routing encapsulation (GRE). RFC 2784, IETF.

[Gross et al. 2018] Gross, J., Ganga, I., and Sridhar, T. (2018). Geneve: Generic network virtualization encapsulation. Internet Draft, Work in progress, IETF.

[Halpern and Pignataro 2015] Halpern, J. and Pignataro, C. (2015). Service function chaining (SFC) architecture. RFC 7665, IETF.

[Jouet et al. 2015] Jouet, S., Cziva, R., and Pezaros, D. P. (2015). Arbitrary packet matching in openflow. In *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6.

[Kicinski and Viljoen 2016] Kicinski, J. and Viljoen, N. (2016). ebpf hardware offload to smartnics: cls bpf and xdp. *Proceedings of netdev*, 1.

[Mahalingam et al. 2014] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and Wright, C. (2014). Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. RFC 7348, IETF.

[McCanne and Jacobson 1993] McCanne, S. and Jacobson, V. (1993). The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA. USENIX Association.

[Miano et al. 2018] Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M. V., and Tumolo, M. (2018). Creating Complex Network Services with eBPF: Experience and Lessons Learned. *High Performance Switching and Routing (HPSR). IEEE*, pages 1–8.

[Mijumbi et al. 2016] Mijumbi, R., Serrat, J., Gorricho, J. L., Bouten, N., De Turck, F., and Boutaba, R. (2016). Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys and Tutorials*, 18(1):236–262.

[Orebaugh et al. 2006] Orebaugh, A., Ramirez, G., and Beale, J. (2006). *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier.

[Pacífico et al. 2018] Pacífico, R. D., Coelho, G. R., Vieira, M. A., and Nacif, J. A. (2018). Roteador sdn em hardware independente de protocolo com análise, casamento e ações dinâmicas. In *Simpósio Brasileiro de Redes de Computadores (SBRC)*, volume 36.

[Pfaff et al. 2015] Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. (2015). The design and implementation of open vswitch. In *Proceedings of the 12th USENIX*

Conference on Networked Systems Design and Implementation, NSDI'15, pages 117–130, Berkeley, CA, USA. USENIX Association.

[Qazi et al. 2013] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. (2013). SIMPLE-fying middlebox policy enforcement using SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):27–38.

[Quinn et al. 2018] Quinn, P., Elzur, U., and Pignataro, C. (2018). Network service header (NSH). RFC 8300, IETF.

[Quinn and Nadeau 2015] Quinn, P. and Nadeau, T. (2015). Problem statement for service function chaining. RFC 7498, IETF.

[Tu et al. 2017] Tu, C., Stringer, J., and Pettit, J. (2017). Building an extensible open vswitch datapath. *SIGOPS Oper. Syst. Rev.*, 51(1):72–77.

[Xhonneux et al. 2018] Xhonneux, M., Duchene, F., and Bonaventure, O. (2018). Leveraging ebpf for programmable network functions with ipv6 segment routing. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 67–72, New York, NY, USA. ACM.