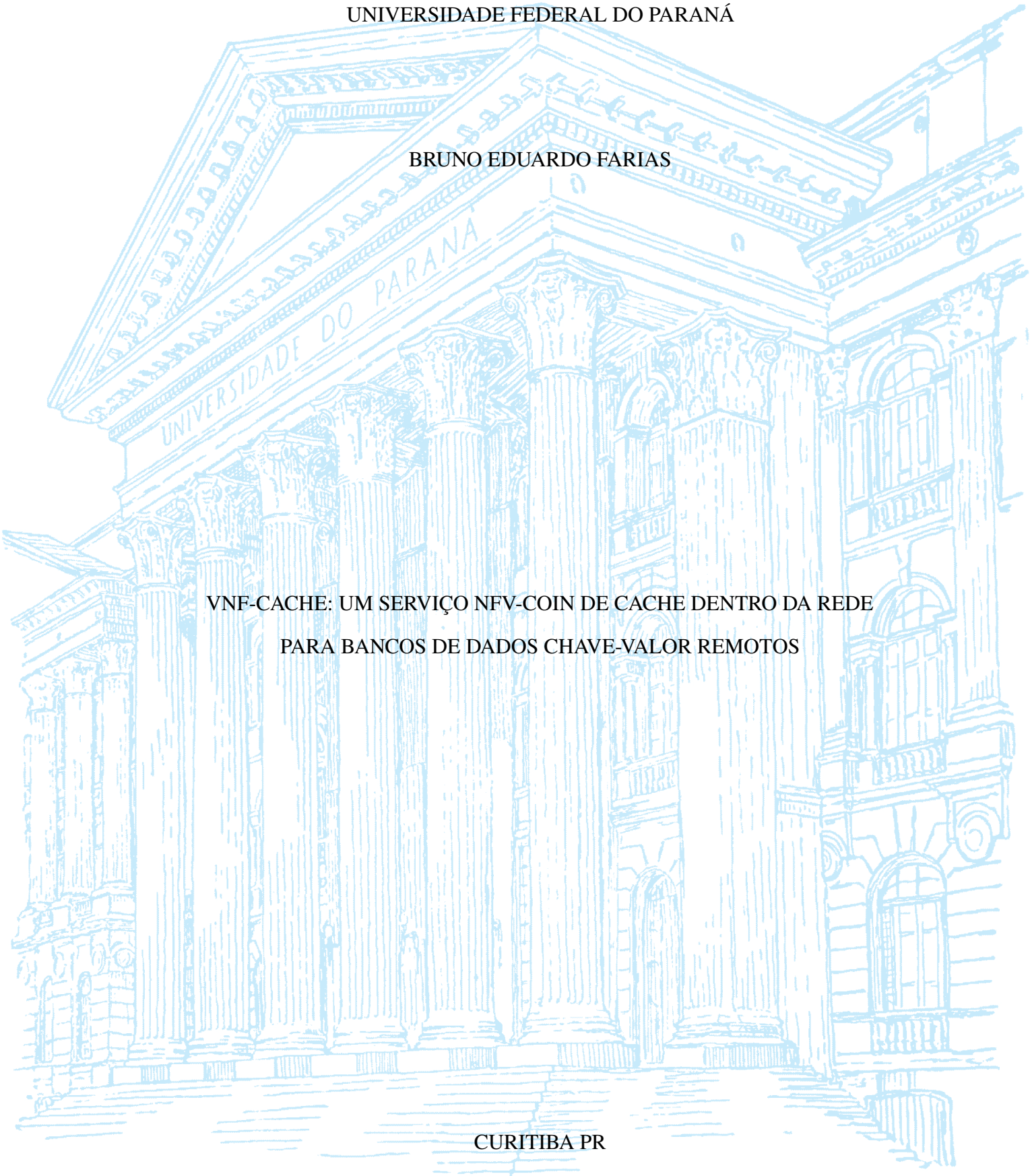
UNIVERSIDADE FEDERAL DO PARANÁ

BRUNO EDUARDO FARIAS



VNF-CACHE: UM SERVIÇO NFV-COIN DE CACHE DENTRO DA REDE PARA BANCOS DE DADOS CHAVE-VALOR REMOTOS

CURITIBA PR

2023

BRUNO EDUARDO FARIAS

VNF-CACHE: UM SERVIÇO NFV-COIN DE CACHE DENTRO DA REDE PARA BANCOS DE DADOS CHAVE-VALOR REMOTOS

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*

Orientador: Elias Procópio Duarte Júnior

Coorientador: José Flauzino

CURITIBA PR

2023

AGRADECIMENTOS

Agradeço primeiramente a Deus por me permitir e me guiar durante esse período de aprendizado, sem sua ajuda nada disso teria sido possível.

À minha noiva Isabella, que sempre esteve ao meu lado me dando apoio, amor e cuidado. Sua compreensão, encorajamento e paciência foram essenciais para essa conquista.

Aos meus pais e família por todos os cuidados e conselhos a mim dedicados. Seu amor incomparável foi meu combustível para perseverar perante as dificuldades.

Ao meu orientador Elias e coorientador José pelas sábias orientações e pela dedicação ao sucesso dessa pesquisa.

RESUMO

Com o crescimento exponencial da quantidade de dados disponíveis na Internet, é cada vez mais necessário otimizar o tempo de resposta das requisições e o uso dos recursos de rede e computação envolvidos no acesso aos dados. Neste contexto, umas das principais soluções utilizadas são as caches, uma tecnologia que permite diminuir drasticamente o tempo de acesso de requisições por dados. Seu princípio de funcionamento é baseado no armazenamento das informações frequentemente acessadas em localizações mais próximas aos solicitantes, dispensando a necessidade de requisições repetitivas aos servidores. Esta monografia apresenta a VNF-Cache, um serviço de cache para bancos de dados do tipo chave-valor localizados geograficamente distantes dos solicitantes. A VNF-Cache é baseada em Virtualização de Funções de Rede (*Network Function Virtualization* - NFV), uma tecnologia que possibilita a implementação de funções de rede em software, viabilizando sua execução em infraestruturas virtualizadas sobre máquinas com hardware de propósito geral. Desta forma, a tecnologia NFV reduz a dependência por hardware especializado, possibilitando a redução dos custos de implementação e de operação das funções de rede, além de permitir maior flexibilidade no gerenciamento dos sistemas. Por sua vez, bancos de dados chave-valor são bancos de dados não-relacionais que realizam a persistência dos dados através da associação de uma única chave para cada dado armazenado. Os principais objetivos da VNF-Cache são reduzir o tempo de resposta das requisições, o tráfego de rede realizado e a utilização dos recursos computacionais. Para isso, a VNF-Cache se baseia no conceito de NFV-COIN, que possibilita a implementação de serviços arbitrários e inovadores diretamente na rede, através do paradigma *Computing In The Network* (COIN). O funcionamento básico da VNF-Cache consiste na aproximação das informações aos clientes, armazenando os dados mais requisitados diretamente na rede e em locais geograficamente mais próximos. Por este motivo, a VNF-Cache deve estar localizada no caminho entre o cliente e o servidor remoto, realizando a interceptação e o processamento dos pacotes de rede trafegados. Através deste processamento, é possível armazenar diretamente na VNF-Cache os valores das chaves requisitadas pelos clientes, de forma que caso uma chave requisitada esteja válida na cache, seu valor é retornado diretamente para o cliente, dispensando a necessidade de reencaminhar os pacotes para o servidor. Através de uma implementação para prova de conceito e dos experimentos realizados com servidores geograficamente distantes, foi obtida uma redução significativa no tempo de reposta e um aumento na quantidade de requisições processadas por segundo.

Palavras-chave: Cache; *COmputing In the Network*; Virtualização de Funções de Rede; Bancos de Dados Chave-Valor

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

API	*Application Programming Interface*
AWS	*Amazon Web Services*
BD	Banco de Dados
BSON	*Binary JavaScript Object Notation*
CAPEX	*Capital Expenditure*
CDN	*Content Delivery Network*
COIN	*Computing In The Network*
CPU	*Central Processing Unit*
DRAM	*Dynamic Random Access Memory*
ETSI	*European Telecommunications Standards Institute*
FIFO	*First In, First Out*
GB	*Gigabyte*
HD	*Hard Disk*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
INC	*In Network Computing*
IoV	*Internet of Vehicles*
JSON	*JavaScript Object Notation*
KVM	*Kernel-based Virtual Machine*
KVS	*Key-Value Store*
LFU	*Least Frequently Used*
LRU	*Least Recently Used*
MANO	*Management and Orchestration*
NFV	*Network Function Virtualization*
NFVI	*Network Function Virtualization Infrastructure*
NFVO	*Network Function Virtualization Orchestrator*
NFVS	*Network Function Virtualization Supervisor*
OPEX	*Operational Expenditure*
RAM	*Random Access Memory*
SAND	*Server and Network Assisted Dynamic Adaptative Streaming over HTTP*
SO	Sistema Operacional
SFC	*Service Function Chain*
SRAM	*Static Random Access Memory*
UFPR	Universidade Federal do Paraná

VM	*Virtual Machine*
VNF	*Virtualized Network Function*
VNFM	*Virtualized Network Function Manager*
VIM	*Virtualized Infrastructure Manager*
VPN	*Virtual Private Network*

SUMÁRIO

1 INTRODUÇÃO

Apesar de seu surgimento ser datado do início da década de 1960, as caches ainda são uma das principais tecnologias utilizadas para a otimização do tráfego de dados. No contexto de sua criação, as implementações de caches eram exclusivamente em hardware, tendo como objetivo o armazenamento dos dados frequentemente acessados em um tipo de memória de rápido acesso localizada mais próxima ao processador (Smith, 1982). Com isso, a necessidade de se realizar consultas na memória principal reduz consideravelmente, já que os dados frequentemente requisitados podem ser encontrados na cache. Além disso, o tempo de busca por algum dado em memória também é reduzido, já que o tempo de acesso das memórias cache é significativamente menor do que o das memórias principais. Embora o uso de caches no contexto de hardware ainda seja muito comum, as suas aplicações se expandiram para outros contextos, como os de Navegadores *Web* e de Sistemas Operacionais. Desta forma, se aproveitando da localização dos dados, as caches também podem ser implementadas em software, expandindo ainda mais as possibilidades de aplicação.

Esta monografia propõe a VNF-Cache, um serviço de cache para bancos de dados chave-valor baseado na tecnologia de Virtualização de Funções de Rede (*Network Function Virtualization* - NFV). A NFV possibilita a implementação de funções de rede em software que pode ser executado em infraestruturas virtualizadas (ETSI, 2012). Com isso, as tradicionais funções de rede, como roteadores e *firewalls*, que são normalmente executadas em hardware dedicado, podem ser executadas em máquinas de propósito geral. Desta forma, é possível reduzir os custos de capital (*CAPital EXpenditures* - CAPEX) e de operação (*OPerational EXpenditures* - OPEX), já que as funções de rede não são mais dependentes do hardware específico e podem ser executadas em máquinas de menor custo de aquisição e de operação (Venâncio et al., 2022). Além disso, outro benefício é a maior flexibilidade e facilidade de gerenciamento dos serviços de rede (Flauzino e Jr., 2022).

Como a gama de funções de rede que podem ser implementadas utilizando NFV é muito variada, a arquitetura de referência NFV-MANO (NFV - *MANagement and Orchestration*) surgiu para padronizar as implementações e possibilitar a interoperabilidade entre os sistemas NFV (ETSI, 2021). Para isto, a arquitetura define o gerenciamento do ciclo de vida das *Virtualized Network Functions* (VNFs) e dos recursos computacionais por elas utilizados (Mijumbi et al., 2016). A arquitetura NFV-MANO é composta basicamente por três blocos fundamentais e interdependentes: (i) o *Virtualized Infrastructure Manager* (VIM), responsável por gerenciar a infraestrutura e os recursos computacionais disponibilizados para a virtualização, (ii) o NFV *Orchestrator* (NFVO), responsável pelo gerenciamento do ciclo de vida dos serviços de rede e da infraestrutura de virtualização entre os múltiplos VIMs e (iii) o VNF *Manager* (VNFM), responsável pelo gerenciamento do ciclo de vida das múltiplas VNFs em execução.

Recentemente, foi proposta a arquitetura NFV-COIN (NFV - *COmputing In the Network*) para possibilitar a implementação de serviços arbitrários e inovadores diretamente na rede, através do paradigma chamado *COmputing In the Network* (COIN). Para Venâncio et al. (2022), a arquitetura NFV-COIN deve permitir que os próprios usuários finais realizem a implantação dos serviços na rede. Desta forma, além de se aproveitar dos benefícios do paradigma NFV, como flexibilidade e redução de custos, a arquitetura NFV-COIN também abre novas possibilidades de aplicações, já que suas implementações devem ser genéricas o suficiente para possibilitar a compatibilidade entre as diversas linguagens de programação e técnicas de virtualização existentes.

A VNF-Cache tem como propósito agilizar o acesso de clientes a bancos de dados chave-valor geograficamente distantes. Com a globalização crescente das últimas décadas, é comum encontrarmos informações sendo frequentemente requisitadas em diferentes partes do mundo. Isto ainda é incentivado pelas tendências e pelas informações que estão em alta no momento. Ou seja, se um determinado dado se torna popular globalmente, o número de requisições para ele tende a aumentar de maneira exponencial e repentina. Desta forma, o número de acessos aos registros dos bancos de dados é cada vez maior, assim como a necessidade de maximizar a usabilidade das aplicações pelos usuários. Por este motivo, um dos grandes desafios dos dias de hoje é garantir a agilidade e a fluidez do uso da Internet. Para isso, é necessário reduzir e estabilizar o tempo de resposta das requisições e o uso do processamento computacional disponibilizado, evitando gargalos que possam prejudicar a experiência dos usuários.

Neste contexto, um dos principais recursos utilizados para gerenciamento e armazenamento de informações são os bancos de dados. Por sua vez, o *Key-Value Store* (KVS), ou banco de dados chave-valor, é um tipo de banco de dados não-relacional que realiza a persistência dos dados através da associação de uma única chave para cada dado armazenado (Seeger, 2009). O uso deste tipo de banco de dados permite ao desenvolvedor da aplicação o armazenamento dos dados sem o uso de esquemas, ou seja, sem o tradicional método relacional de linhas e colunas pré-definidas. Desta forma, a flexibilidade no projeto do banco de dados é maior, assim como a qualidade do código de programação correspondente.

Para desempenhar a funcionalidade de *caching* dos conjuntos chave-valor, a VNF-Cache deve estar localizada em algum ponto da rede entre os clientes e os servidores de banco de dados chave-valor. Seu funcionamento é através da filtragem dos pacotes de rede e do armazenamento dos valores das chaves requisitadas pelos clientes. Por este motivo, o tráfego dos pacotes deve ser desviado para a VNF-Cache, que irá realizar o gerenciamento interno de acordo com a operação carregada por cada pacote filtrado. Se um dado requisitado está em cache, ele é retornado diretamente para o cliente e a VNF-Cache não redireciona o pacote para o servidor. Por outro lado, se o dado não estiver em cache, a VNF-Cache redireciona o pacote para o servidor e, se possível, armazena o valor correspondente após a resposta.

Uma arquitetura da VNF-Cache é proposta, composta por três módulos: (i) o VNF-Cache *Filter* que é responsável por realizar a filtragem dos pacotes de acordo com os fluxos pré-definidos, (ii) o VNF-Cache *Manager*, que realiza o gerenciamento do destino dos pacotes e das alterações internas da VNF-Cache (conforme as operações de manipulação de dados contidas nos pacotes), e (iii) o VNF-Cache *Storage*, o responsável pelo armazenamento dos conjuntos chave-valor da cache.

Como prova de conceito, foi realizada uma implementação da VNF-Cache utilizando a linguagem de programação Python[1], o banco de dados MongoDB[2] e a biblioteca *PyMongo*[3] para comunicação com o banco de dados. Embora o *MongoDB* não seja por padrão um banco de dados chave-valor, é possível adaptar sua utilização para realizar requisições por chaves, simulando um comportamento chave-valor. Além disso, é possível adaptar a VNF-Cache para funcionamento com outros bancos de dados explicitamente chave-valor, como o *Redis*, por exemplo. Nesta implementação, o fluxo dos pacotes PyMongo dos clientes é desviado para uma porta específica da instância em que a VNF-Cache está executando. Após o processamento do cabeçalho dos pacotes, a VNF-Cache pode tomar a decisão adequada, como redirecionar o pacote para o servidor e aguardar pela resposta, retornar o valor diretamente para o cliente, fazer alterações internas no VNF-Cache *Storage*, entre outros.

[1]https://docs.python.org/3/reference/

[2]https://www.mongodb.com/docs/manual/reference/

[3]https://pymongo.readthedocs.io/en/stable/api/index.html

Através da avaliação empírica, foram efetuados experimentos com três cenários diferentes: (A) cliente, VNF-Cache e servidor de banco de dados chave-valor próximos entre si, (B) cliente e VNF-Cache próximos entre si, distanciando o servidor e (C) cliente, VNF-Cache e servidor distantes entre si. Os resultados destes experimentos apontam que a utilização de uma VNF-Cache no cenário A não é eficiente, podendo até mesmo apresentar piora no tempo de resposta das requisições e da quantidade de requisições processadas por segundo. Por outro lado, os experimentos B e C apontam que em um cenário de longas distâncias entre clientes e servidores, a utilização de uma VNF-Cache é capaz de reduzir consideravelmente o tempo de resposta das requisições e aumentar o número de requisições processadas por segundo.

O restante deste trabalho está organizado da seguinte maneira. O Capítulo 2 descreve o conceito de cache, desde a sua origem no contexto de hardware até as atuais aplicações em software. O Capítulo 3 apresenta uma visão geral da tecnologia NFV e das arquiteturas NFV-MANO e NFV-COIN, bem como trabalhos relacionados. O Capítulo 4 apresenta a proposta da VNF-Cache, a arquitetura, a implementação do protótipo para prova de conceito e resultados experimentais. Por fim, o Capítulo 5 relata as conclusões e os trabalhos futuros decorrentes deste.

2 SERVIÇO DE CACHE

Com o crescimento exponencial dos sistemas de informação e comunicação, são cada vez mais necessárias estratégias para diminuir o tempo de resposta de consultas de dados e otimizar o uso do processamento computacional disponibilizado. Em um mundo globalizado e movido por tendências, a disparada de acessos pelos mesmos conjuntos de dados faz com que o número de requisições em porções semelhantes de conteúdo seja alto. Consequentemente, o tempo de resposta é prejudicado. Além disso, essas requisições repetitivas podem utilizar grandes quantidades de processamento e de banda de rede.

Embora sua origem seja datada do início da década de 1960, as memórias caches são hoje uma das principais abordagens utilizadas na computação para otimizar o acesso aos dados nas arquiteturas modernas de computadores (Patterson e Hennessy, 2014). Este capítulo tem como objetivo definir o conceito de cache e dissertar sobre a sua eficiência em resolver o problema de acesso a dados frequentemente requisitados. São discutidas definições de localidade, seu funcionamento básico, suas principais aplicações e suas políticas de acesso e escrita para manter a coerência de dados.

2.1 DEFINIÇÕES BÁSICAS

Segundo Smith (1982), memórias cache são pequenos *buffers* de memória de alta velocidade utilizados em sistemas de computação modernos para armazenar temporariamente porções de conteúdo da memória principal. Estes *buffers* são posicionados estrategicamente próximos aos processos que estão realizando requisições destes conteúdos. Seus principais objetivos são acelerar o acesso aos dados e evitar a sobrecarga de trabalho da memória principal. Como exemplificado por Smith (1982, p.1):

> "Por exemplo, em alguns dos principais computadores (como Amdahl 470V/7, IBM 3033), a memória principal pode ser acessada entre 300 a 600 nanossegundos; por outro lado, as informações podem ser obtidas de uma cache entre 50 a 100 nanossegundos".

Já Patterson e Hennessy (2014) definem cache como qualquer armazenamento utilizado para tomar vantagem da localização do acesso. Em suma, todas as requisições de dados para a memória principal passam primeiramente pela cache. Se o dado buscado for encontrado na cache, seu valor é imediatamente retornado ao cliente solicitante. Caso contrário, a busca segue para os níveis mais baixos da hierarquia de memória (mais detalhes na Seção 2.2.1). Um dos conceitos mais importantes relacionados à utilização de memória cache é a localidade, que pode ser tanto temporal quanto espacial, definidas a seguir.

2.1.1 Localidade Temporal

Para Handy (1993), a *localidade temporal* se refere ao fato de que as instruções que são executas em sequência próxima umas das outras possuem uma alta probabilidade de acessarem os mesmos dados de forma frequente. Este fato beneficia a cache ao facilitar o armazenamento dos dados que são frequentemente utilizados. Por outro lado, a execução de instruções espalhadas ao longo do tempo requisitando os mesmos dados acabam por dificultar a escolha do que manter em cache e o que invalidar para liberar espaço para novas entradas. Além disso, a localidade

temporal também é percebida no fato de os processos normalmente seguirem um padrão de repetição de acesso e escrita dos mesmos dados durante um período, através de laços e fluxos de execução. “Se um item foi referenciado, então ele tende a ser referenciado novamente em breve” (Patterson e Hennessy, 2014, p.374).

2.1.2 Localidade Espacial

Já a *localidade espacial* é definida como a probabilidade de que se um item de memória de uma posição específica for requisitado, os itens localizados em endereços de memória próximos também serão requisitados em algum momento próximo (Patterson e Hennessy, 2014). Isto pode ser observado em códigos de programação nos quais são percorridas estruturas de dados como vetores e matrizes, já que na grande maioria das vezes estes dados são armazenados em posições contínuas de memória. Porém, Handy (1993) afirma que mesmo que as posições de memória não estejam em uma única faixa de endereços e sim significativamente espalhadas ao longo do espaço de endereçamento da memória, o acesso aos dados da memória ainda pode ser beneficiado pela uso de uma cache.

2.2 FUNCIONAMENTO BÁSICO

Uma vez que seu objetivo principal é reduzir a latência de acesso aos dados, as caches normalmente ficam localizadas próximas das aplicações que fazem as requisições. Ainda no contexto de memória cache em arquitetura de computadores, aproveitando-se dos conceitos de localidade temporal e espacial, as caches buscam armazenar os dados que são mais frequentemente requisitados em um tipo de memória mais eficiente e cujo tempo de acesso aos dados seja consideravelmente menor. Para isso, um conceito essencial relacionado é o de hierarquia de memória e seus principais tipos, descritos a seguir.

2.2.1 Hierarquia de Memória

Para Patterson e Hennessy (2014), uma hierarquia de memória consiste de múltiplos níveis de memória com diferentes tamanhos e tipos. Este modo de implementação de sistema de memória é comumente utilizado para contornar os limites de capacidade e custo dos diferentes tipos de memória existentes. A Tabela 2.1 resume as principais tecnologias existentes, suas latências médias de acesso e o custo médio por GB (gigabyte), em dólares.

Tabela 2.1: Tipos de Memória com tempo médio de acesso e custo por GB em 2012.

Tecnologia	**Tempo médio de acesso**	**Custo por GB ($)**
SRAM	0.5 - 2.5 ns	$500-$1000
DRAM	50 - 70 ns	$5-$10
Flash	5.000 - 50.000 ns	$0.75-$1
HD	5.000.000 - 20.000.000 ns	$0.05-$0.10

Embora nos dias de hoje o custo médio por GB tenha diminuído consideravelmente, os tipos de memória de acesso rápido como as *Static Random Access Memory* (SRAM) ainda possuem um custo maior do que as memórias de acesso mais lento como os *Hard Disks* (HD). Desta maneira, no topo da hierarquia ficam memórias de acesso rápido, porém de pequena capacidade. Por sua vez, na parte inferior da hierarquia ficam memórias de acesso lento e de maior capacidade, que podem ultrapassar as dezenas de terabytes.

Deste modo, o fluxo das requisições por dados flui das aplicações, passando pelas memórias de alta hierarquia até as de baixa hierarquia e, se necessário, acessando o HD. Como as caches ficam localizadas próximas das aplicações, elas são consultadas primeiramente e, se os dados estiverem válidos, a busca termina e o dado é retornado para o processador.

Neste modelo de implementação de sistema de memórias, existe um fluxo constante de dados entre os diferentes níveis da hierarquia. Se um dado começa a ser muito requisitado, ele passa a ser armazenado em níveis mais altos da hierarquia. Por outro lado, se um dado nos altos níveis passa a ser pouco solicitado ele desce de nível na hierarquia, liberando espaço para o armazenamento de outros dados. A Figura 2.1 ilustra a ordem hierárquica dos níveis de memória, bem como o sentido de aumento da latência de acesso e da capacidade de armazenamento.

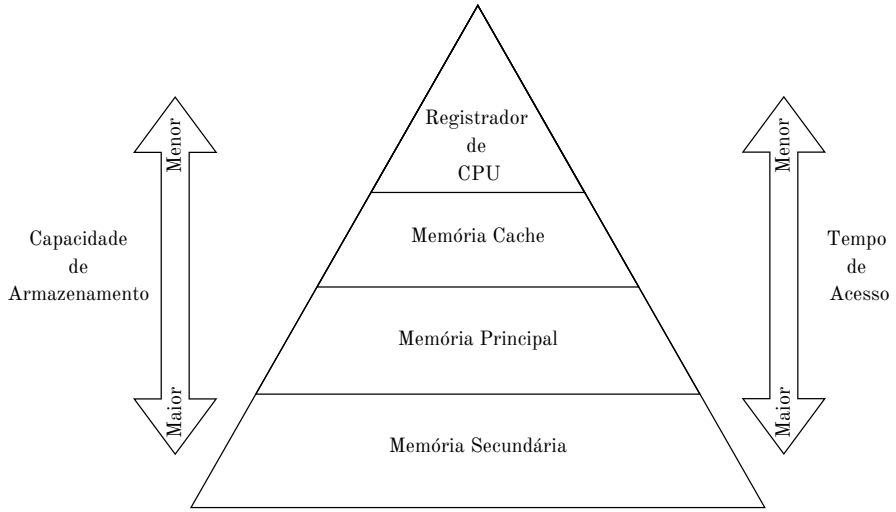


Figura 2.1: Hierarquia de memória com sentido de crescimento da capacidade e do tempo de acesso.

2.2.2 Cache *Hit* e *Miss*

Durante o processo de busca por um dado na memória, a cache é requisitada. Caso esse dado esteja presente na cache, dizemos que ocorreu um cache *hit* (Patterson e Hennessy, 2014). Neste caso, o valor é retornado diretamente para o solicitante, sem prosseguir para os demais níveis abaixo na hierarquia. Desta forma, não existe a latência característica dos tipos de memória de baixo nível, podendo reduzir de forma significativa o tempo de resposta dependendo do correto posicionamento e preenchimento das caches.

Por outro lado, quando o dado requisitado não está presente na cache, dizemos que ocorreu um cache *miss*, e a busca prossegue para o nível diretamente abaixo (Patterson e Hennessy, 2014). Caso o dado esteja presente neste outro nível, o dado é retornado e a busca termina. Caso o dado não esteja presente, a busca segue para os demais níveis, aumentando a latência total da busca.

Baseado nessas definições, é possível perceber a necessidade de realizar um gerenciamento adequado de quais dados devem ser mantidos em cache (para aumentar a quantidade de cache *hits*). Manter uma cache preenchida com dados pouco requisitados pode fazer com que o desempenho da requisição seja inferior, já que ocorre apenas a adição de um *overhead* na busca e a maioria das requisições estão resultando em cache *miss*. Neste contexto, são utilizadas políticas

de preenchimento que são detalhadas na Seção 2.4, que descreve estratégias de preenchimento de cache para resolver estes problemas.

2.2.3 Armazenamento dos Dados na Cache

Existem várias maneiras de implementar o armazenamento dos dados na cache. Para Patterson e Hennessy (2014), o modo mais simples de armazenar dados em uma cache é utilizando o chamado mapeamento direto. Neste modo, cada dado da memória principal é referenciado de maneira única em uma tabela na cache utilizando o seu endereço da memória principal. Desta forma, durante a busca por uma informação, basta comparar se o endereço buscado na memória principal está referenciado na cache. Além do endereço, outros campos auxiliares também são utilizados na busca, como as *tags*, que contêm mais informações sobre o endereço e o bit de validade, que indica se aquela entrada ainda é válida ou já expirou.

Além do mapeamento direto, existem outras opções para o gerenciamento do armazenamento, variando de acordo com a associatividade (relacionamento entre linhas de cache e os dados), tamanho da cache e complexidade do hardware. Em seu livro *The Cache Memory Book* (Handy, 1993), o autor disserta sobre os diferentes tipos de implementação de cache, detalhando com exemplos cada modo.

2.3 APLICAÇÕES

Embora as caches tenham surgido no contexto de arquitetura de computadores, atualmente elas não se limitam a este cenário e são encontradas em diversos outros contextos em sistemas computacionais. As técnicas tradicionais de implementação de caches em hardware deixaram de ser o único foco, gerando oportunidades para implementações de caches em software, ampliando ainda mais as possíveis aplicações na computação. A seguir, alguns exemplos de aplicações são descritos.

2.3.1 Navegadores Web

Os navegadores web são exemplos clássicos de aplicação do conceito de cache para otimização do acesso a dados. Páginas web e documentos que são frequentemente acessados pelos usuários são armazenados pelo navegador na memória local, possibilitando uma redução do tempo de carregamento das páginas nos acessos seguintes.

Essa forma de implementação é diferente do modelo apresentado anteriormente, pois caso o dado procurado não esteja válido na cache, ao invés de realizar a busca em diferentes níveis de hierarquia de memória, a requisição é encaminhada para o servidor que hospeda as páginas daquele site. Neste caso, quando ocorre um cache *hit*, temos uma grande queda na taxa de uso de banda de rede, já que ao invés de encaminhar cada requisição necessária para construir uma página *HyperText Markup Language* (HTML) para o servidor, os arquivos já são prontamente disponibilizados pela cache.

Outro uso comum para caches em navegadores são as caches de *cookies*. Segundo Juels et al. (2006), é comum servidores introduzirem *cookies* contendo valores secretos nos navegadores com o objetivo de personalizar as páginas web de acordo com o perfil do usuário. Além disso, os *cookies* podem ser utilizados para tornar a autenticação de serviços mais prática e segura.

2.3.2 Sistemas Operacionais

Outro exemplo clássico de aplicação do conceito de cache são os Sistemas Operacionais (SO). Estes utilizam sistemas de cache em software para manter cópias de arquivos frequentemente utilizados, tendo como objetivo obter um carregamento mais rápido e otimizado (Jacob et al., 2008). O *Microsoft Windows*, por exemplo, implementa um sistema de gerenciamento de cache baseado em um conjunto de funções do *kernel* (núcleo central do SO). Estas funções cooperam com o gerenciador de memória para providenciar *caching* de dados para todos os *drivers* dos sistemas de arquivos, incluindo os locais e os de rede. Além disso, os autores comentam sobre as outras funções do gerenciador de cache, como por exemplo:

- Decidir quais partes dos arquivos são mantidas em cache e quais são mantidas em memória secundária;
- Manter a coerência da cache (ver Seção 2.4);
- Gerenciar a cópia dos dados entre a cache, a memória principal e a memória secundária;

De forma semelhante, os sistemas baseados em *Unix* também implementam caches nativamente. No Linux, por exemplo, há uma cache com o objetivo de manter na memória principal os dados mais úteis do HD (Jacob et al., 2008).

2.3.3 Caches em *Proxy*

As caches em *proxy* também são utilizadas para manter cópias de dados e páginas da web, porém em pontos fixos, que costumam ser mais distantes do usuário final. Enquanto as caches de navegadores ficam diretamente nas máquinas dos usuários, as caches em *proxy* se encontram no caminho entre as máquinas dos usuários (os clientes, neste caso) e o servidor remoto, especificamente na borda da rede. A Figura 2.2 mostra o posicionamento de uma cache em *proxy* em uma rede.

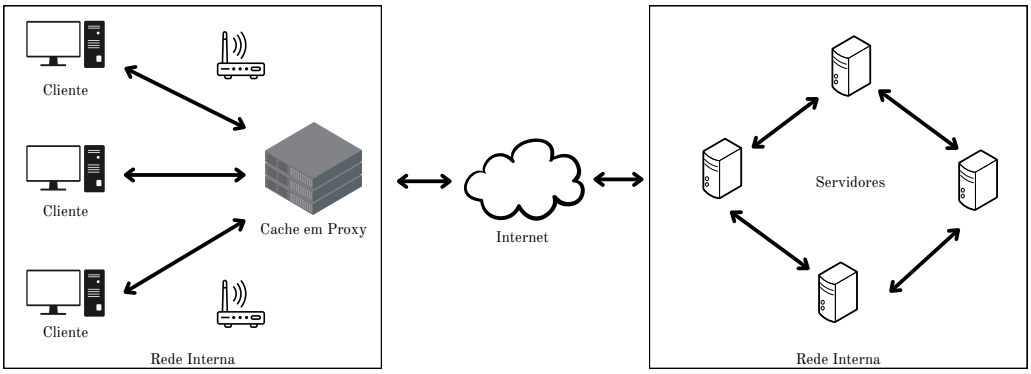


Figura 2.2: Localização de uma Cache em *Proxy* na rede.

Estas caches normalmente são mais úteis nos primeiros acessos aos dados. Isto ocorre pois os navegadores normalmente implementam caches com boas políticas de preenchimento e substituição, como as descritas na Seção 2.4. Desta forma, as requisições repetitivas serão respondidas pela cache do navegador e o fluxo da busca é interrompido antes mesmo de chegar na cache em *proxy*. Segundo (Amiri et al., 2003, p.376):

> "A execução na borda não apenas aumenta a escalabilidade do *back-end*, mas ainda reduz a latência de resposta para o cliente e evita o superprovisionamento dos recursos do *back-end* pelo fato de os recursos da borda serem compartilhados entre os sites".

2.4 COERÊNCIA E POLÍTICAS DE PREENCHIMENTO E SUBSTITUIÇÃO DE DADOS

Manter a coerência é um dos principais desafios encontrados nas implementações de cache, tanto em hardware quanto em software. Manter cópias dos dados frequentemente acessados pode ser desafiador, principalmente se estes mesmos dados são alterados com frequência na origem e o fornecimento de informação desatualizada pode invalidar todo o contexto da busca realizada. Segundo Patterson e Hennessy (2014, p.466): "Informalmente, é possível dizer que um sistema de memória é coerente se qualquer leitura de um dado retorna o valor mais recente escrito para aquele dado". Para manter a coerência, são utilizadas algumas políticas de preenchimento e substituição de dados na cache, descritas a seguir.

2.4.1 Políticas de Tratamento de Escritas

Para tratar eventos de escrita de dados na memória e manter a coerência entre as informações da cache e as da origem, são utilizadas políticas de tratamento como *Write-Through*, *Write-Back* e *Write-Invalidate*, descritos a seguir. Se uma atualização for realizada somente na cache ou somente na origem, os dados podem se tornar incoerentes e seu uso em requisições posteriores ser impossibilitado.

A técnica do *Write-Through* consiste em simplesmente atualizar os dados tanto na cache quanto na origem (Jacob et al., 2008). Desta maneira, é possível garantir que os dados serão idênticos nas duas memórias e a consistência será mantida. Porém, essa técnica pode prejudicar o desempenho, já que toda escrita deve ser atualizada na origem, aumentando consequentemente a latência final e o tráfego de dados entre as memórias (Patterson e Hennessy, 2014). Por este motivo, a técnica deve ser aplicada com maior frequência em sistemas nos quais a taxa de escrita dos dados é baixa, ou seja, os dados permanecem constantes ao longo do tempo. Neste caso, as poucas escritas são realizadas em ambas as memórias, e a latência de escrita na origem não influencia o desempenho da cache de maneira significativa. Além disso, Smith (1982) afirma que sistemas computacionais com múltiplos processadores podem utilizar a técnica de *Write-Through* para utilizar a memória principal como um armazenamento compartilhado e consistente de informações entre os processos.

Por outro lado, a técnica de *Write-Back* busca postergar as escritas na origem, realizando temporariamente as escritas apenas na cache. Como definido por Patterson e Hennessy (2014), a atualização dos dados nos níveis mais baixos da hierarquia (ou na origem) acontece somente quando o dado é substituído na cache. Ou seja, os valores mais recentes dos dados ficam somente em cache até que sejam sobrescritos por outros, só então são escritos na origem. Essa técnica pode reduzir consideravelmente a latência e o tráfego de dados em comparação com a técnica de *Write-Through*, já que a quantidade de escritas realizadas na origem é reduzida. Porém, é necessário implementar um gerenciamento mais complexo da cache, já que diferentes versões de um mesmo dado podem estar presentes no sistema ao mesmo tempo e o risco de fornecer informações incorretas aumenta consideravelmente (Handy, 1993) e Patterson e Hennessy (2014).

Por fim, a técnica de *Write-Invalidate* torna os dados da cache inválidos e realiza a atualização apenas na origem. Desta forma, nas próximas requisições de um dado para a cache, a mesma é forçada a buscar os dados novamente na origem, que terá o valor mais atualizado (Jacob et al., 2008). Esta técnica facilita a implementação da cache, porém a latência no tempo

de resposta tende a ser maior. Isto ocorre pois na próxima requisição por este dado o mesmo não estará em cache, ou seja, um (cache *miss*) ocorrerá, e será necessário aguardar a consulta na origem, que possui um tempo de acesso maior do que a cache.

2.4.2 Políticas de Preenchimento e Substituição de Dados da Cache

Para obter um bom desempenho da cache, é necessário um bom gerenciamento de quais dados manter em cache e quais manter apenas em memória principal. Para isso, existem políticas pré-definidas de gerenciamento, como *Least Recently Used* (LRU), *Least Frequently Used* (LFU), *First-In, First-Out* (FIFO) e até mesmo políticas aleatórias.

A política *Least Recently Used* (LRU) (Utilizado Menos Recentemente, em tradução livre) é uma política de substituição de dados que retira da cache os dados que foram requisitados há mais tempo, ou seja, que não foram utilizados recentemente. Para Einziger et al. (2018), a LRU é baseada na hipótese de que o item utilizado menos recentemente também é o item menos provável de ser utilizado no futuro. Desta forma, ao remover esse item da cache, podemos inserir novos dados que foram recentemente requisitados pelos processos.

De forma semelhante ao LRU, a *Least Frequently Used* (LFU) (Utilizado Menos Frequentemente, em tradução livre) retira da cache os dados que são menos frequentemente utilizados. Para isso, o gerenciador da cache mantém um contador para o número de requisições realizadas para cada item da cache. Por conta dessa necessidade de monitoramento dos itens, a LFU produz um *overhead* e deve ser feito um bom gerenciamento do processamento utilizado (Einziger et al., 2018).

A política *First-In, First-Out* (FIFO) (Primeiro a Entrar, Primeiro a Sair, em tradução livre) por sua vez pode ser associada ao conceito de uma fila. Nessa política, os itens da cache mantém um *timestamp* do horário de entrada e, quando ocorre a necessidade de retirar algum item para alocar outro, o gerenciador da cache seleciona o mais antigo para ser sobrescrito. Este tipo de implementação deve ser utilizado com precaução, pois o item mais antigo da cache não necessariamente é o item menos utilizado.

Por fim, a política *Random* (aleatória) de substituição de dados não segue um padrão na troca de informações. Nessa política, quando um item precisa ser retirado da cache, o gerenciador da cache escolhe aleatoriamente qual item vai ser retirado. Para Handy (1993, p.59):

> "Uma das alternativas mais populares para um LRU é o algoritmo de substituição aleatória. [...] A base do algoritmo é que o item a ser substituído é escolhido de forma aleatória. A implementação é simples [...]".

2.5 CONCLUSÕES PARCIAIS

Neste capítulo foi apresentado o conceito de cache e outros conceitos básicos relacionados à sua implementação. O objetivo principal de uma cache é reduzir a latência de acesso aos dados da memória principal. Foi descrito o seu funcionamento básico, a sua relação com a hierarquia de memória, além dos conceitos de cache *hit* e *miss*, aplicações, localidade e políticas de coerência.

3 VIRTUALIZAÇÃO DE FUNÇÕES DE REDE & NFV-COIN

Este capítulo tem como objetivo definir o que é a *Network Function Virtualization* (NFV), ou Virtualização de Funções de Rede, explicando o conceito, seus principais benefícios e as arquiteturas NFV-MANO (NFV - *Management and Orchestration*), para padronização das implementações NFV, e NFV-COIN, para implantação de serviços de *COmputing In the Network* (COIN) utilizando tecnologias NFV.

3.1 VIRTUALIZAÇÃO DE FUNÇÕES DE REDE

A Virtualização de Funções de Redes, ou NFV, é uma alternativa concreta para implementar serviços de redes utilizando técnicas de virtualização, como *Virtual Machines* (VMs) e *containers*, que podem ser executadas em máquinas com hardware comum, como as de prateleira. Assim como as implementações em hardware, o uso de *Virtualized Network Functions* (VNFs) também possibilita a implementação de uma variedade de tipos diferentes de serviços de rede, como roteadores, *Virtual Private Network* (VPN), funções de análise de tráfego, *firewalls*, *Content Delivery Network* (CDN), entre outros (ETSI, 2012).

De acordo com Flauzino e Jr. (2022), conforme citado por (Martins et al., 2014), através da aplicação desta técnica, é possível reduzir os custos de capital (*CAPital EXpenditures* - CAPEX) e de operação (*OPerational EXpenditures* - OPEX), já que a função de rede não é mais dependente do hardware dedicado e pode ser executada em máquinas de menor custo de aquisição e de operação. Além disso, existe uma maior flexibilidade na composição e no gerenciamento dos serviços de rede, já que a instanciação ou a remoção dos mesmos é realizada de forma mais rápida e prática. Ainda, o redimensionamento da infraestrutura de hardware da virtualização, aumentando ou diminuindo recursos de acordo com a demanda pode ser realizado de forma mais simples e de acordo as necessidades de cada função de rede. Ao contrário dos equipamentos de rede em hardware, que são produzidos por um número reduzido de fabricantes, as funções e serviços virtualizados de rede têm o potencial de permitir um número muito maior de desenvolvedores, que podem disponibilizar seus produtos através de *marketplaces* na própria Internet (Bondan et al., 2019).

O conceito de virtualização de funções de rede surgiu em meados de 2012 como fruto de pesquisa realizada em organizações de telecomunicações como a americana AT&T, a espanhola Telefonica e a italiana Telecom Italia (Mijumbi et al., 2016). Segundo a *European Telecommunications Standards Institute* (ETSI, 2012, p. 5), a tecnologia NFV:

> "[...] envolve a implementação de funções de rede em software que podem ser executadas em hardware de uma variedade de servidores, e que pode ser movido ou instanciado em várias localizações na rede conforme necessário, sem a necessidade de instalação de novos equipamentos."

Desde a especificação da tecnologia em 2012, vários pesquisadores de diversas organizações se uniram para estabelecer padrões e desenvolver arquiteturas para gerenciar e padronizar a implantação das funções de rede virtualizadas. Um dos principais resultados destes esforços é a arquitetura de referência *MANagement and Orchestration* (MANO), que será descrita a seguir.

3.1.1 A Arquitetura MANO

Como a gama de funções de rede que podem ser implementadas utilizando NFV é muito variada, surge a necessidade de desenvolver uma arquitetura para padronizar a configuração e a implantação das mesmas. Proposta originalmente em 2014 pela mesma organização criadora do NFV (a ETSI), a arquitetura de referência NFV-MANO surgiu para auxiliar a resolução destes desafios, bem como permitir a interoperabilidade entre os sistemas NFV.

Para isto, a arquitetura define o gerenciamento do ciclo de vida das VNFs e dos recursos computacionais utilizados por elas (Mijumbi et al., 2016). A arquitetura NFV-MANO é composta basicamente por três blocos fundamentais e interdependentes: o *Virtualized Infrastructure Manager* (VIM), o NFV *Orchestrator* (NFVO) e o VNF *Manager* (VNFM) (ETSI, 2021), que são apresentados a seguir. A Figura 3.1 mostra os relacionamentos entre o NFV-MANO, a *Network Functions Virtualization Insfraestructure* (NFVI) e as VNFs.

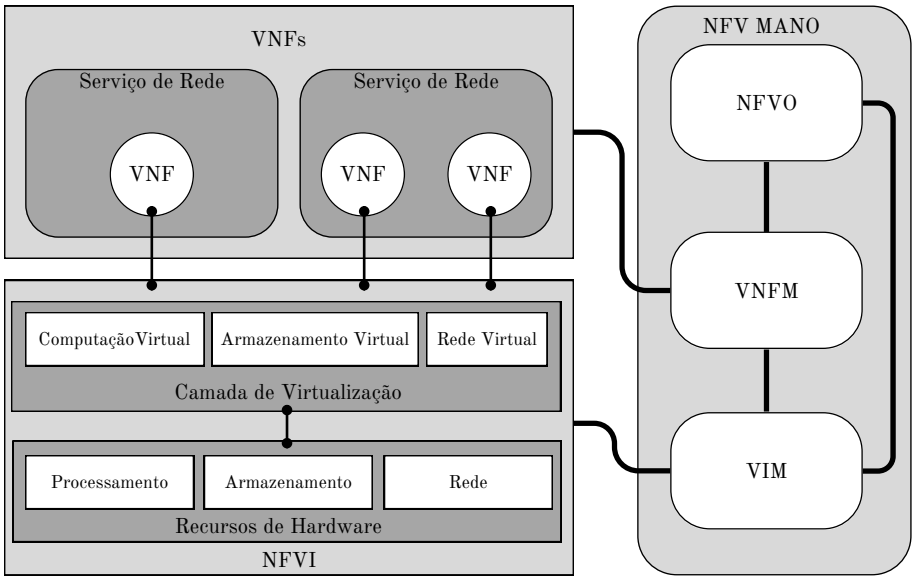


Figura 3.1: Arquitetura MANO e relacionamento com as VNFs e serviços de rede.

Os VIMs são responsáveis por controlar e gerenciar a infraestrutura e os recursos disponíveis para a virtualização. Um único VIM pode ser responsável por gerenciar apenas um tipo de recurso computacional por vez (como o processamento, a memória ou a rede), ou vários tipos simultaneamente (ETSI, 2021). Este deve ser capaz de distribuir eficientemente os recursos disponibilizados para uma NFVI (ou Infraestrutura de Virtualização de Funções de Rede, em tradução livre), de forma a atender às necessidades de cada VNF sem impactar a execução das demais. A NFVI é o conjunto do hardware físico juntamente com a camada de virtualização que executa as VNFs.

Já o NFVO possui duas funções principais: gerenciar a NFVI entre os múltiplos VIMs existentes e coordenar o ciclo de vida dos serviços de rede. Além disso, o NFVO possibilita a composição de VNFs para a criação de outros serviços mais complexos, formando uma *Service Function Chain* (SFC) (Fulber-Garcia et al., 2020b). As múltiplas VNFs podem ser organizadas entre si para formar SFCs com topologias diversas (Fulber-Garcia et al., 2020a). Além disso, podem ser orquestradas de forma a garantirem propriedades específicas, como tolerância a falhas (Venâncio e Duarte Jr, 2022). É possível, inclusive, conectar funções de rede em múltiplos domínios distintos baseadas em orquestradores heterogêneos (Huff et al., 2020).

Por sua vez, o VNFM é responsável pelo gerenciamento do ciclo de vida de instâncias de VNF, ou seja, este módulo é encarregado de instanciar e remover instâncias, realizar configurações

e atualizações, além de monitorar e escalonar as funções (Venâncio et al., 2021a). Um único VNFM é responsável por gerenciar diversas funções de rede virtualizadas de diferentes tipos (ETSI, 2021).

Em resumo, as funções de redes virtualizadas são gerenciadas pelos VNFMs e são executadas em infraestruturas de virtualização de funções de redes (os NFVIs). Estas, por sua vez, são gerenciadas pelos VIMs, que são integrados e gerenciados pelo NFVO. Este deve distribuir todos os recursos computacionais entre os VIMs e fazer o gerenciamento dos demais serviços de rede necessários. Os sistemas NFV-MANO, como o Vines (Flauzino e Duarte Jr, 2022), incluem todos os componentes descritos acima, mas cada VNF deve ser executada através de uma plataforma de execução de VNF (Tavares et al., 2018; Garcia et al., 2019).

3.2 NFV-COIN

Conforme apresentado anteriormente, a NFV possibilita a implementação de uma variedade de funções de rede, desde as mais simples até os serviços mais complexos, formados pela composição de múltiplas funções. Neste sentido, Venâncio et al. (2022) propuseram o NFV-COIN, uma arquitetura para possibilitar a implementação de serviços arbitrários e inovadores diretamente na rede, através do paradigma chamado *COmputing In the Network* (COIN). Um termo alternativo para o mesmo paradigma é o *In-Network Computing* (INC), que tem sido usado no contexto da implementação de serviços na rede utilizando hardware (Sapio et al., 2017). Diversos serviços já foram implementados com a tecnologia NFV-COIN, entre eles: detectores de falhas (Turchetti e Duarte, 2015; Turchetti e Duarte Jr, 2017), consenso (Venâncio et al., 2021b) e a difusão confiável e ordenada (Venâncio et al., 2019).

A NFV-COIN apresenta algumas das mesmas vantagens das aplicações em hardware, sendo a flexibilidade das implementações a mais notável. Além disso, a arquitetura NFV-COIN apresenta outras vantagens, como a redução de custos (tanto de implementação quanto de manutenção), de consumo de energia, de espaço físico utilizado e do uso de recursos computacionais. Porém, os autores ressaltam que, devido aos desafios presentes ao mover os serviços para a rede, as versões implementadas em hardware, por possuírem recursos especializados, comumente apresentam um melhor desempenho. A Figura 3.2 representa visualmente a composição e a integração da arquitetura NFV-COIN.

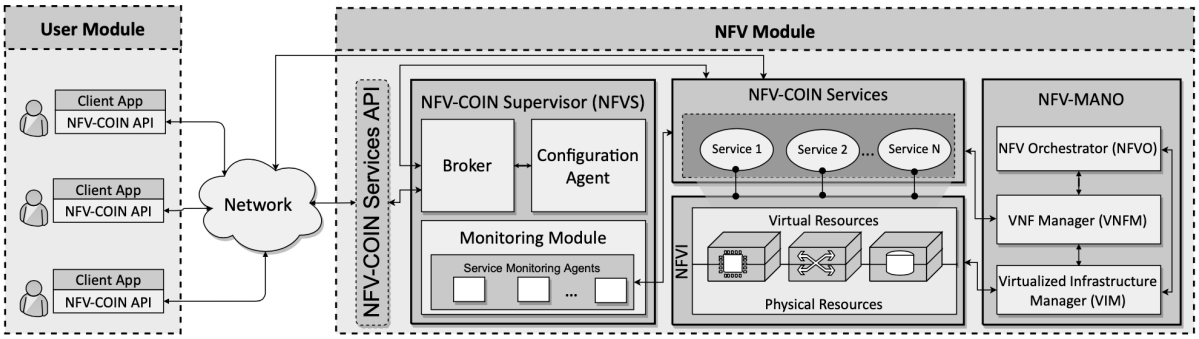


Figura 3.2: Uma Arquitetura NFV-COIN. Fonte: (Venâncio et al., 2022).

Para Venâncio et al. (2022), a arquitetura NFV-COIN deve ser capaz de ser utilizada por usuários finais para realizar a implantação de serviços na rede através do uso de uma *Application Program Interface* (API). Esta deve ser implementada em um módulo que é executado diretamente na máquina do usuário, provendo primitivas padronizadas para o gerenciamento dos

serviços NFV-COIN, como a criação, a configuração e a exclusão. Além disso, a arquitetura deve implementar um NFV *Module*, que é composto por: (i) a NFV-COIN *Services* API, que é uma API responsável por tratar as requisições entre os usuários e os serviços NFV-COIN, (ii) por um sistema tradicional de NFV (NFV-MANO, NFVs e NFVI) e (iii) por um NFV-COIN *Supervisor* (NFVS), que é um ponto de acesso para os serviços NFV-COIN.

Além disso, os autores também definem que a API de serviços deve ser genérica o suficiente para permitir o acesso pelos usuários finais de maneira simples aos serviços NFV-COIN. Isto deve ser realizado através de implementações com primitivas padronizadas, que possibilitem o uso de diferentes técnicas de virtualização e que possuam compatibilidade com a maioria das linguagens de programação, a fim de obter a interoperabilidade entre os desenvolvedores (Venâncio et al., 2022).

Em resumo, o NFV-COIN é uma arquitetura que possibilita a implantação de serviços diretamente na rede, através do uso de APIs padronizadas que possam ser utilizadas pelos usuários finais. Além disso, ela pode ser descrita como uma extensão da arquitetura MANO, com a integração de uma API para o usuário final, de outra API de serviços e de um componente NFVS para gerenciamento, configuração e monitoramento dos serviços COIN, além dos próprios serviços NFV-COIN.

3.3 TRABALHOS RELACIONADOS UNINDO NFV E CACHES

Nos últimos anos, vários trabalhos foram desenvolvidos envolvendo a união de NFV e de caches. Zhuang et al. (2019), por exemplo, discutem sobre a possibilidade de se aplicar caches baseadas em NFV para minimizar o tempo de recuperação de conteúdos em sistemas de *Internet-of-Vehicles* (IoV). Para os autores, o uso deste tipo de cache pode facilitar a implantação dos serviços e a disseminação de seus conteúdos, possibilitando uma melhor confiabilidade e eficiência dos serviços IoV.

Já Clayman et al. (2018) propõem uma arquitetura para *streaming* de vídeo *Server and Network Assisted Dynamic Adaptative Streaming over HTTP* (SAND). Nesta arquitetura, instâncias de caches virtualizadas são criadas conforme a demanda por conteúdo. Além disso, os autores também discorrem sobre os posicionamentos dessas instâncias no grafo da rede, baseando-se em características como a largura de banda dos caminhos, os locais e o número de clientes da rede.

Outro trabalho relevante é o de Liu et al. (2017), que discutem o ganho de desempenho notável que a aplicação de caches NFV em redes 5G pode causar. Além da flexibilidade, dinamicidade e escalabilidade possibilitadas pelo uso de NFV, os autores ainda destacam a possibilidade de oferecer serviços de cache para provedores de serviços e para operadoras de rede utilizando a mesma infraestrutura. Para Liu et al. (2017):

> "Nosso objetivo é melhorar a eficiência da distribuição de conteúdos, possibilitando o cache oportunístico e/ou o pré-carregamento de conteúdo em diferentes pontos da infraestrutura de rede virtualizada."

Além destes, vários outros trabalhos são relevantes na área, como Zheng et al. (2018), Veitch et al. (2017) e Shih et al. (2016).

3.4 CONCLUSÕES PARCIAIS

Neste capítulo foi apresentado o conceito de NFV, descrevendo seus principais benefícios, as arquiteturas NFV-MANO e NFV-COIN, além de trabalhos relacionados envolvendo a união

de caches e NFV. Com a NFV, podemos reduzir a dependência de hardware dedicado através da execução de funções de rede em ambientes virtualizados, implementando-as em software e executando-as em hardware de propósito geral. Esta estratégia aumenta a flexibilidade da composição dos serviços de rede, reduz os custos de implementação e de operação, além de facilitar o gerenciamento dos sistemas. Através da arquitetura NFV-MANO, é possível padronizar as implementações de VNFs. Por fim, a arquitetura NFV-COIN, através do uso de APIs, de um sistema NFV e de módulos auxiliares, possibilita a execução, por usuário finais, de serviços arbitrários diretamente na rede, através do paradigma COIN.

4 UM SERVIÇO DE CACHE NA REDE PARA BANCOS DE DADOS CHAVE-VALOR

Com base nas definições dos capítulos anteriores, este capítulo tem como objetivo propor a VNF-Cache, uma implementação de um serviço de cache como uma função de rede virtualizada para servidores de banco de dados chave-valor. Será apresentado o seu conceito, uma arquitetura para implantação, seu funcionamento, uma implementação para prova de conceito e resultados experimentais.

4.1 APLICANDO NFV-COIN PARA CACHE NA REDE DE BANCO DE DADOS CHAVE-VALOR

Um dos modelos de armazenamento em banco de dados mais utilizado atualmente é o *Key-Value Store* (KVS), ou Banco de Dados Chave-Valor, em tradução livre. Estes são bancos de dados não-relacionais que realizam a persistência dos dados através da associação de uma única chave para cada dado armazenado. Segundo Seeger (2009), os bancos de dados chave-valor permitem ao desenvolvedor da aplicação o armazenamento de dados sem esquema, ou seja, sem o tradicional método relacional, no qual são utilizadas tabelas com linhas e com colunas pré-definidas. Nos bancos de dados chave-valor, os conjuntos chave-valor normalmente consistem de uma *string*, que representa a chave, e o dado em si, que é considerado o valor dessa relação chave-valor. Neste caso, os dados não precisam seguir um esquema pré-definido, podendo omitir ou acrescentar informações conforme necessário.

A utilização de bancos de dados chave-valor permite maior flexibilidade no projeto do banco de dados, além de reduzir e melhorar a qualidade do código de programação. Seeger (2009) cita alguns exemplos de bancos de dados chave-valor, como o Apache CouchDB, um banco de dados orientado a documentos, acessível via API, tolerante a falhas e livre de esquema; o Redis, um banco de dados em memória principal, implementado com a linguagem C e que possui foco em desempenho; e o Cassandra, um banco de dados chave-valor desenvolvido pelo Facebook. Além disso, Idreos e Callaghan (2020) citam outras utilizações para bancos de dados chave-valor, como mídias sociais, armazenamento de *logs* de segurança, *caching* de aplicações, processamento de transações online, entre outros.

O acesso a bancos de dados chave-valor remotos é uma necessidade cada vez mais presente. Com a globalização crescente das últimas décadas, é comum encontrarmos informações que são frequentemente requisitadas em diferentes partes do mundo. Isto ainda é incentivado pelas tendências e pelas informações que estão em alta no momento. Ou seja, se algum determinado dado se torna popular globalmente, o número de requisições para ele tende a aumentar de maneira exponencial e repentina. Por isso, é comum requisitarmos informações que estão localizadas em bancos de dados de servidores geograficamente distantes, sejam através dos 340 quilômetros entre as cidades de Curitiba (PR, Brasil) e São Paulo (SP, Brasil) ou os cerca de 18.500 quilômetros entre Curitiba (PR, Brasil) e Tóquio (Japão), por exemplo. As longas distâncias aumentam a latência total de resposta das requisições, já que os pacotes precisam percorrer longos caminhos na rede global até chegarem aos seus destinos finais.

Neste sentido, a VNF-Cache busca aproximar os dados do cliente, armazenando as informações diretamente na rede e em locais mais próximos, utilizando para isto o conceito de NFV-COIN. A VNF-Cache deve estar localizada no caminho entre o cliente e o servidor remoto, realizando o processamento dos pacotes e armazenando os valores das chaves requisitadas pelos clientes. Caso uma chave requisitada esteja válida nesta cache, seu valor é retornado diretamente

para o cliente, dispensando a necessidade de reencaminhar os pacotes para o servidor. A Figura 4.1 ilustra a sequência de passos do funcionamento da VNF-Cache.

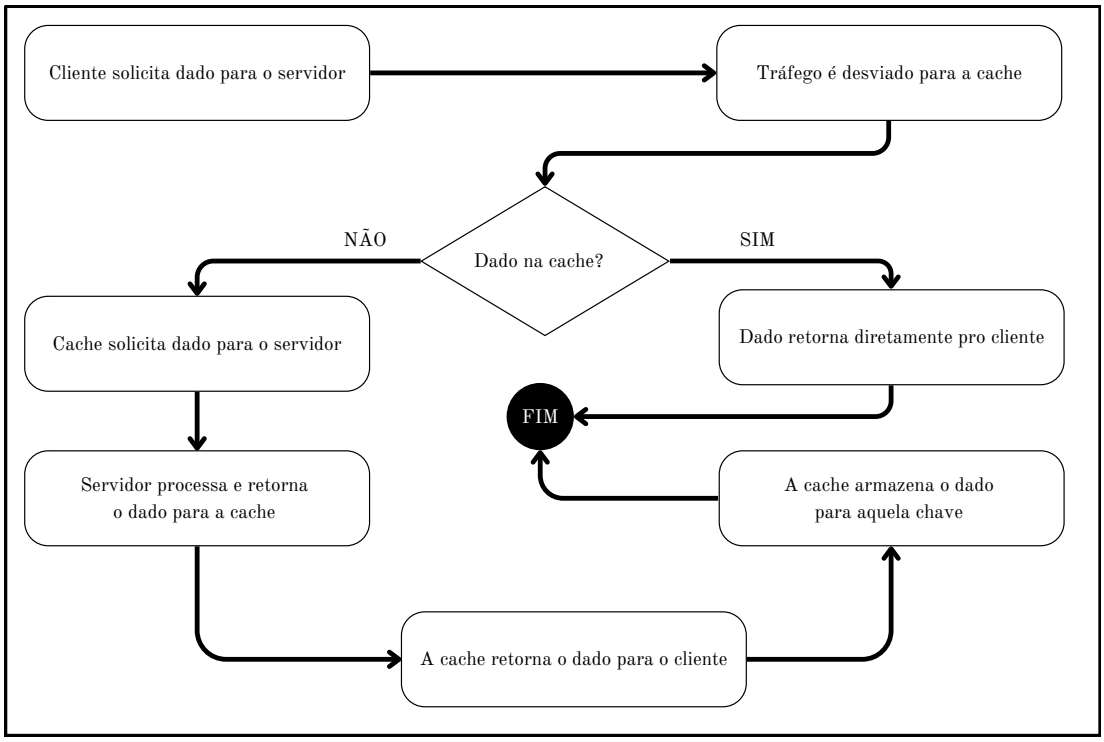


Figura 4.1: Funcionamento da VNF-Cache.

As aplicações que podem se beneficiar da VNF-Cache são diversas, e as soluções podem ser adaptadas ao contexto específico de cada aplicação. A Figura 4.2 mostra uma arquitetura de rede sem a integração de uma VNF-Cache e outra com a integração de uma VNF-Cache, além de mostrar qual o tráfego deduzido e qual o tráfego real. Neste trabalho, o tráfego deduzido é definido como aquele em que o cliente deduz estar causando na rede, que pode ser diferente do tráfego real presenciado na rede. A próxima seção descreve a arquitetura proposta para a VNF-Cache.

4.2 A ARQUITETURA DA VNF-CACHE

É proposta uma arquitetura para a VNF-Cache composta por um conjunto de três módulos interdependentes: um VNF-Cache *Filter*, um VNF-Cache *Manager* e o VNF-Cache *Storage*, ilustrados na Figura 4.3 e descritos a seguir.

O VNF-Cache *Filter*, ou apenas *Filter*, é o módulo responsável pela filtragem dos pacotes recebidos pela VNF-Cache, sejam eles enviados pelos clientes ou pelo servidor. Este módulo é composto por dois outros submódulos: o VNF-Cache *Client Filter* e o VNF-Cache *Server Filter*, que são detalhados a seguir. Em conjunto, estes dois submódulos recebem e filtram os pacotes de rede em três possíveis fluxos: o *Manipulation Flow* (MF), o *Response Flow* (RF) e o *Coordination Flow* (CF), descritos a seguir.

O fluxo de manipulação de dados (MF) é composto pelos pacotes enviados pelos clientes que contenham operações de manipulação de dados, tais como buscas, inserções, atualizações e exclusões. Já o fluxo de respostas RF é composto pelos pacotes enviados pelos servidores e que são respostas para estes pacotes enviados pelos clientes no fluxo MF. Por fim, o fluxo

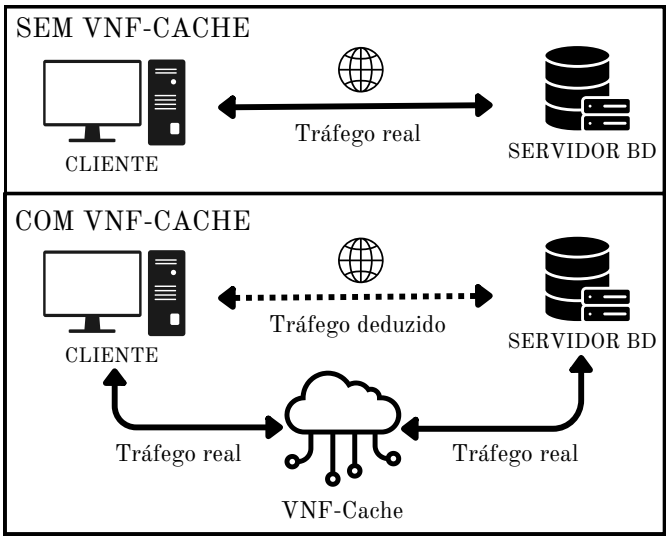


Figura 4.2: Exemplo de arquitetura de rede sem e com VNF-Cache.

de coordenação (CF) é composto pelos demais pacotes trafegados, ou seja, aqueles enviados pelos clientes ou pelos servidores e que possuem outros objetivos, como manter a conexão entre clientes e servidores ou realizar o monitoramento de disponibilidade do servidor.

Desta forma, o *Client Filter* é responsável pela filtragem dos pacotes originados pelo cliente, separando-os entre o fluxo MF, que são os pacotes que comunicam as manipulações de dados solicitadas pelos clientes ao banco, e o fluxo CF, que são os pacotes que realizam as comunicações básicas entre cliente e servidor, neste caso no sentido do cliente para o servidor. De forma semelhante, o *Server Filter* filtra os pacotes vindos do servidor, separando-os entre o fluxo RF, que são os pacotes que respondem as solicitações realizadas pelos clientes no fluxo MF, e o fluxo CF, que neste caso é no sentido do servidor para o cliente.

Por sua vez, o VNF-Cache *Manager*, ou apenas *Manager*, é o principal módulo de gerenciamento da VNF-Cache. De forma semelhante ao *Filter*, o *Manager* também é composto por dois submódulos: o VNF-Cache *Client Manager* e o VNF-Cache *Server Manager*. O *Client Manager* recebe os dois fluxos de pacotes do *Client Filter* e faz o tratamento conforme necessário: o fluxo MF é tratado diretamente com o módulo de armazenamento VNF-Cache *Storage* (que será apresentado a seguir), realizando as leituras, atualizações e exclusões dos dados conforme as operações. Já o fluxo CF é enviado diretamente para a saída da VNF-Cache com destino ao servidor. De forma semelhante, o *Server Manager* recebe os dois fluxos de pacotes do *Server Filter*. Porém, no *Server Manager*, todos os pacotes são enviados para a saída com destino ao cliente, independentemente do fluxo designado pelo *Server Filter*. A diferença entre o processamento dos fluxos no *Server Manager* é que enquanto o fluxo CF é apenas redirecionado para os clientes, os pacotes do fluxo RF passam por um processamento extra, tendo como objetivo o armazenamento dos dados retornados pelo servidor no VNF-Cache *Storage*, conforme as solicitações do *Client Manager*.

Por fim, o VNF-Cache *Storage*, ou simplesmente *Storage*, é o módulo responsável pelo armazenamento das chaves e de seus respectivos valores. Este módulo possui duas funcionalidades principais: (i) retornar o valor de uma chave requisitada pelo *Client Manager* e (ii) armazenar o valor de uma chave capturada pelo *Server Manager* e solicitada pelo *Client Manager*.

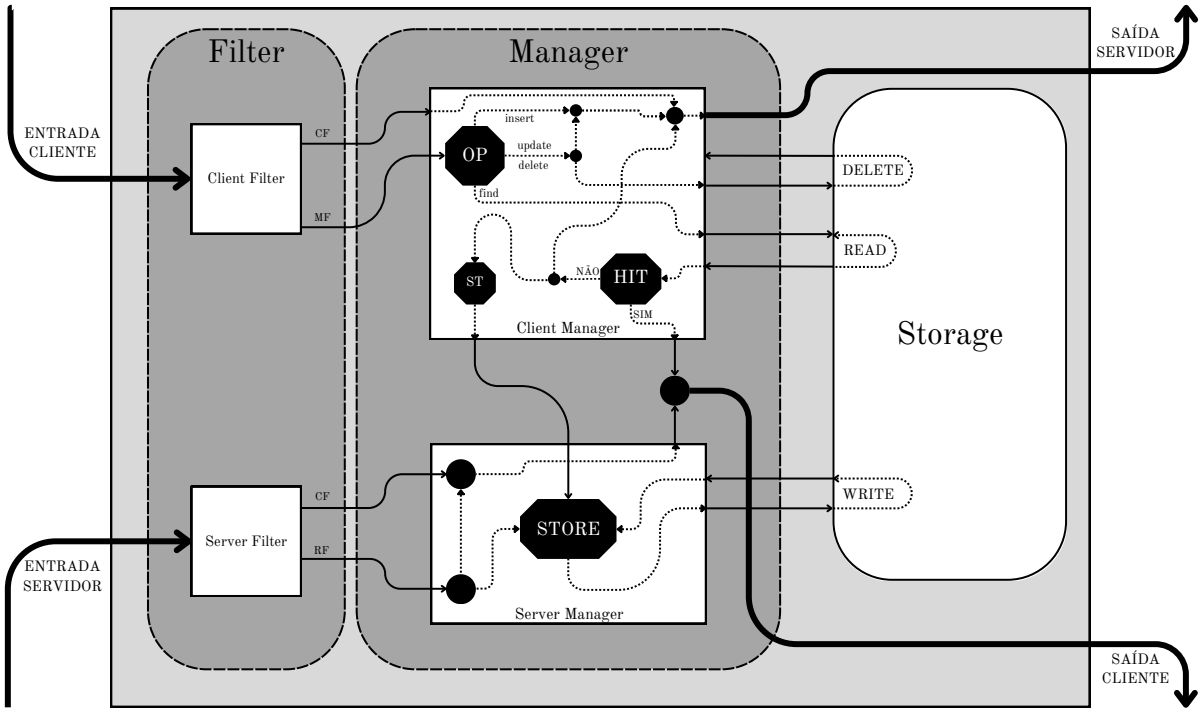


Figura 4.3: Arquitetura da VNF-Cache.

Os dois principais fluxos na arquitetura da VNF-Cache ocorrem após a requisição de consulta de uma chave por algum cliente. Quando um cliente solicita uma chave, o *Client Filter* realiza a filtragem deste pacote no fluxo de manipulação de dados, o MF. Em seguida, o *Client Manager* obtém o tipo de operação, que no caso é *find*, e faz uma requisição ao *Storage* pelo valor daquela chave. Após o retorno do módulo de armazenamento, podemos ter um cache *miss* ou um cache *hit*. Caso a requisição resulte em um cache *hit*, o *Client Manager* retorna o pacote diretamente para o cliente. Por outro lado, caso a requisição seja um cache *miss*, o *Client Manager* redireciona o pacote para o servidor e envia um sinal ao *Server Manager*, alertando-o que a chave requisitada não está na cache e que seu valor deve, na medida do possível, ser armazenado no *Storage* após a resposta do servidor.

4.3 CACHE EM PROXY *VERSUS* VNF-CACHE

Embora sejam conceitos semelhantes, as caches em *proxy* e a VNF-Cache se diferenciam entre si em alguns aspectos, sendo o principal a sua localização na rede. As *proxies* por si só se localizam na borda entre os clientes e a Internet, podendo oferecer diversas funcionalidades, como proteção contra ataques externos, limitação de conteúdo, *firewall*, proteção de dados sensíveis, entre outros (Sysel e Doležal, 2014). Outra funcionalidade das *proxies* são as caches em *proxy*, que fazem o *caching* do conteúdo trafegado entre os clientes e a Internet, ajudando a reduzir consumo de banda, a latência de resposta das requisições e a sobrecarga dos servidores (Cao e Irani, 1997).

Por outro lado, as VNF-Caches podem ser localizadas em diferentes pontos da rede, ao invés de serem fixas na borda da rede entre os clientes e a Internet. Isto possibilita sua inserção no melhor ponto da rede, de acordo com os requisitos e as necessidades dos clientes e/ou servidores. Além disso, as VNF-Caches se aproveitam dos principais benefícios do paradigma NFV, ou seja, a flexibilidade, escalabilidade, facilidade de manutenção e custos reduzidos para implementação.

Outra importante diferença é a forma de processamento dos pacotes para armazenamento nas caches. Na proposta da VNF-Cache, o armazenamento dos dados em cache é através do processamento direto dos pacotes. Ou seja, a VNF-Cache processa cada pacote individualmente para desempenhar sua funcionalidade. Já a cache em *proxy* realiza este processamento em alto nível, fazendo o *caching* de porções de conteúdo, que inclusive podem estar representadas em quantidades variáveis de pacotes de rede.

4.4 IMPLEMENTAÇÃO

Para a implementação da VNF-Cache, algumas decisões tiveram que ser tomadas sobre quais das tecnologias existentes seriam utilizadas. Um ponto importante levado em consideração foi a existência de outros sistemas que realizam o tratamento de pacotes de rede, já que estes poderiam servir de base para a criação da VNF-Cache.

Para o desenvolvimento do módulo da cache foi utilizada a linguagem de programação *Python*. A ampla disponibilidade de bibliotecas para comunicação, análise e tratamento de pacotes de rede é ponto de destaque. Bibliotecas como *Scapy* e *PyShark*, por exemplo, foram fundamentais durante o processo de desenvolvimento.

Além disso, nesta implementação da VNF-Cache foi utilizado o *MongoDB*, um banco de dados orientado a documentos *JavaScript Object Notation* (JSON). Embora este não seja um banco de dados exclusivamente chave-valor, ele pode ser utilizado como tal ao armazenar os dados em forma de documentos flexíveis. Estes, por sua vez, são atrelados a um único índice gerado automaticamente, representando a chave do relacionamento chave-valor. É importante destacar que a VNF-Cache pode ser facilmente adaptada para ser utilizada com outros serviços de bancos de dados chave-valor, como o Redis, por exemplo. Para realizar a comunicação entre os clientes e os servidores de bancos de dados MongoDB, a biblioteca utilizada foi a *PyMongo*, descrita na subseção a seguir.

4.4.1 A Biblioteca PyMongo

A biblioteca PyMongo possui uma interface simples em Python para acesso ao banco de dados MongoDB, conforme mostram os exemplos a seguir.

Para realizar a comunicação entre um cliente e um servidor de banco de dados MongoDB, é necessário estabelecer uma conexão entre eles. Para isso, a biblioteca possui a implementação *MongoClient*, que estabelece uma conexão TCP entre o cliente e o servidor, garantindo a entrega de todos os pacotes na ordem correta. O Algoritmo 4.1 mostra uma conexão entre um cliente e um servidor de banco de dados MongoDB utilizando a biblioteca PyMongo.

Algoritmo 4.1: Conexão ao servidor MongoDB local utilizando PyMongo.

```
1 from pymongo import MongoClient
2
3 client = MongoClient("mongodb://localhost:27017/")
4
5 client.close()
```

Como o MongoDB utiliza os conceitos de bancos de dados e coleções, é necessário selecionar qual coleção de qual banco de dados será utilizada nas operações. Para selecionar o banco de dados, basta utilizar a variável do cliente PyMongo e referenciar o banco de dados, como em *client.nome-do-banco-de-dados*, por exemplo. Já as coleções podem ser acessadas ao referenciar o objeto de banco de dados, como em *nome-do-banco-de-dados["nome-da-coleção"]*.

Assim, para acessar a coleção “filmes” do banco de dados “cinema”, por exemplo, utiliza-se *lista_filmes = client.cinema["filmes"]*.

Para manipular os dados do banco, o MongoDB utiliza as operações tradicionais de leitura, inserção, atualização e exclusão. Porém, isto é realizado através de funções próprias do banco, passando documentos do tipo JSON como parâmetro. Para realizar a leitura dos dados, pode-se utilizar a função *find*, por exemplo. Para realizar a inserção de documentos, podem ser utilizadas funções como *insert_one* ou *insert_many*.

Já para a atualização dos registros do banco de dados, podem ser utilizadas funções como *update_one* ou *update_many*. Por fim, para realizar a exclusão de dados, podem ser utilizadas as funções *delete_one* e *delete_many*. Além destas funções, outras funções podem ser utilizadas para manipulação dos dados, conforme estão descritas na documentação do MongoDB[1] e do PyMongo[2]. O Algoritmo 4.2 mostra um exemplo de inserção e de busca de um documento utilizando a biblioteca.

Algoritmo 4.2: Inclusão e Busca de documento no MongoDB utilizando PyMongo.

```
1  from pymongo import MongoClient
2
3  client = MongoClient("mongodb://localhost:27017/")
4
5  db = client["cinema"]
6  collection = db["filmes"]
7
8  novo_filme = {
9     "nome": "Perdido em Marte",
10    "ano": 2015,
11    "resumo": "O astronauta Mark Watney realiza uma
12       missão para Marte, [...]"
13 }
14
15 #realiza a inserção do dado no banco de dados
16 collection.insert_one(novo_filme)
17
18 #realiza a busca e impressão do documento
19 doc = collection.find({"nome": "Perdido em Marte"})
20 print(doc)
21
22 client.close()
```

Para estabelecer esta comunicação entre o cliente e o servidor MongoDB, a biblioteca PyMongo utiliza pacotes de rede com um cabeçalho pré-definido, que é brevemente apresentado na seção a seguir.

4.4.2 Tratamento dos Pacotes Gerados pela Biblioteca PyMongo

Para realizar a comunicação dos clientes com o servidor de banco de dados MongoDB, a biblioteca PyMongo utiliza pacotes de rede com o protocolo de rede IP e transporte TCP. Como a variedade de recursos do MongoDB é grande, a biblioteca possui diversas funções para se comunicar com o banco MongoDB. Porém, neste trabalho o foco é apenas no pacote com identificador de operação 2013, que é o tipo padrão para requisições de busca, inserção, exclusão e alteração do PyMongo. Os pacotes do PyMongo possuem um cabeçalho padrão com 25 bytes

[1] https://www.mongodb.com/docs/manual/reference/

[2] https://pymongo.readthedocs.io/en/stable/index.html

de tamanho, sendo separados em 7 campos, que estão ilustrados na Figura 4.4 e que são descritos a seguir.

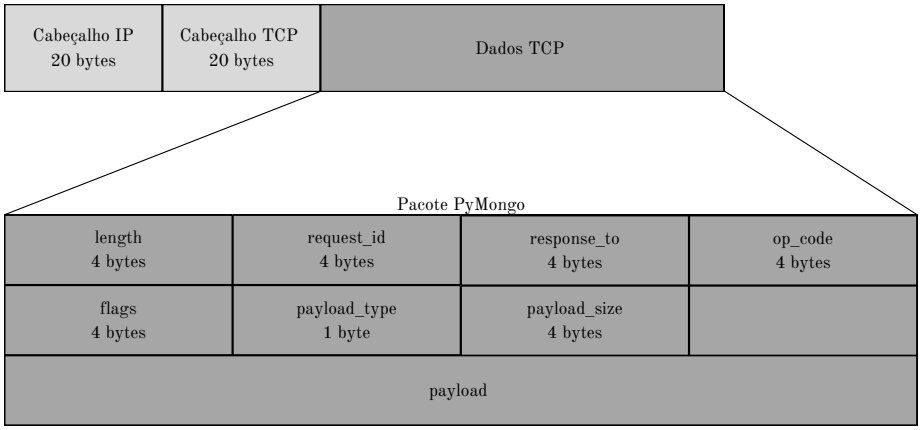


Figura 4.4: Cabeçalho de um pacote PyMongo.

O primeiro campo (4 bytes) é o ***length***, que contém o tamanho total do pacote TCP. O segundo campo (4 bytes) é o ***request_id***, que contém um identificador único da requisição (este identificador deve estar presente no pacote de resposta). O terceiro campo (4 bytes) é o ***response_to***, que contém o identificador ao qual aquele pacote se refere (caso este seja a resposta para algum outro pacote enviado anteriormente). O quarto campo (4 bytes) é o ***op_code***, que contém o número de identificação da operação (neste caso focamos apenas na operação de código 2013). O quinto campo (4 bytes) contém algumas ***flags*** para comunicação entre PyMongo e MongoDB. O sexto campo (1 byte) é o ***payload_type*** e contém o tipo do conteúdo do pacote PyMongo. Por fim, o sétimo campo (4 bytes) é o ***payload_size***, que contém o tamanho total do documento *Binary JSON* (BSON) contido naquele pacote.

Ao unirmos o conteúdo em binário do ***payload_size*** e o restante do *payload* do pacote, podemos obter o JSON completo que foi enviado pelo PyMongo utilizando a classe ***RawBSONDocument*** da biblioteca ***bson***. Desta forma, foram viabilizados os devidos tratamentos dos pacotes dentro da implementação da VNF-Cache, facilitando e tornando eficiente o monitoramento e a filtragem dos pacotes que são de fato importantes. Através dos campos do cabeçalho do pacote PyMongo, a VNF-Cache pode tomar as decisões corretas para cada evento, como reenviar o pacote para o servidor e armazenar a resposta, retornar o valor armazenado diretamente para o cliente, aplicar as políticas de preenchimento e substituição, etc.

4.4.3 Implementação das Funcionalidades de Serviço de Cache

Na implementação da VNF-Cache, a cache propriamente dita é armazenada em um único arquivo Python, unindo todos os módulos da arquitetura em um só. Seu funcionamento é exatamente como o proposto anteriormente, ou seja, o tráfego de pacotes de rede com destino ao servidor de banco de dados é desviado para uma porta específica da cache. Esta, por sua vez, analisa os pacotes recebidos e faz os devidos tratamentos de acordo com a necessidade de cada requisição. Este desvio dos pacotes de rede pode ser implementado de acordo com a infraestrutura na qual a VNF-Cache está sendo implantada. Uma possível maneira de realizar este redirecionamento é através da adição de políticas de roteamento diretamente no plano de controle da rede. Desta forma, ao trafegar durante o percurso na rede, os pacotes com destino ao

servidor de banco de dados são redirecionados para a VNF que está executando a VNF-Cache. A seguir será apresentado o funcionamento da cache desde o recebimento do pacote até seu envio para o cliente ou para o servidor.

4.4.3.1 Sockets e Threading

Para realizar o monitoramento dos pacotes recebidos pela cache, um *socket* da biblioteca padrão do Python é aberto na porta especificada e aguarda por requisições de estabelecimento de conexão pelos clientes. Quando este *socket* recebe um pedido de conexão de algum cliente, uma nova *thread* é criada. Desta forma, é possível tratar de forma simultânea as requisições de cada cliente, reduzindo assim o tempo de tratamento dos pacotes e, consequentemente, o tempo de resposta para cada cliente.

Em cada *thread* aberta, um novo *socket* é criado para estabelecer a comunicação direta entre a VNF-Cache e o servidor MongoDB. Desta forma, para cada cliente que se comunica com o servidor de banco de dados, existe na VNF-Cache uma *thread* com dois *sockets* ativos, um tratando as comunicações entre o cliente e a cache, e o outro realizando as comunicações entre a cache e o servidor de banco de dados MongoDB. Após o estabelecimento das conexões entre cliente e cache, e cache e servidor, a VNF-Cache aguarda pelos pacotes que serão enviados pelo cliente.

Para realizar o papel do módulo VNF-Cache *Filter*, quando um pacote é recebido na VNF-Cache, é feita uma leitura do cabeçalho PyMongo e seus campos são separados de acordo com os tamanhos pré-definidos. Como relatado na Seção 4.4.2, o foco da cache está nos pacotes do fluxo MF, ou seja, os pacotes que possuam o campo *op_code* igual a 2013 e que contenham operações de busca, inserção, atualização e exclusão. Caso o código da operação não seja 2013 (fluxo de coordenação), como por exemplo nos pacotes de estabelecimento de conexão do PyMongo, de estatística e de monitoramento de disponibilidade, os pacotes são reenviados diretamente para o servidor MongoDB através do *socket* estabelecido anteriormente. Da mesma forma, os pacotes do fluxo de coordenação recebidos do servidor também são reenviados diretamente para o cliente, sem alteração nenhuma tanto na cache quanto nos pacotes.

Por outro lado, caso o código da operação do pacote seja 2013 (fluxo de manipulação), a VNF-Cache realiza uma decodificação do pacote, identificando a operação de manipulação e em quais dados a manipulação será executada. É necessário encontrar qual método está sendo utilizado, qual a chave sendo requisitada e, se for a primeira requisição de uma chave, qual é o valor retornado pelo servidor para aquela chave. Para isso, é realizada a reconstrução do documento JSON que foi enviado pelo PyMongo, utilizando-se da combinação dos códigos binários do tamanho do documento, que é disponibilizado no cabeçalho PyMongo, e do conteúdo do pacote, que fica logo após o fim do cabeçalho PyMongo. Após a reconstrução do documento JSON, é obtida uma estrutura de dados que pode ser facilmente manipulada através de métodos padrão para verificar a existência de chaves definidas e realizar o acesso os respectivos valores.

4.4.3.2 Análise do JSON

Após o JSON original ser reconstruído, o VNF-Cache *Manager* pode realizar o processamento dos pacotes que contém operações de buscas, inserções, alterações ou exclusões. O código a seguir mostra um exemplo de um JSON que faz a requisição de um dado pela chave *McDonalds* na coleção *reviews* do banco de dados *restaurants*.

Algoritmo 4.3: Exemplo de JSON do PyMongo para busca de dados pela chave *McDonalds*.

```
1 {
```

```
2     "find": "reviews",
3     "filter": {
4        "name": {
5           "$eq": "McDonalds"
6        }
7     },
8     "sort": {
9        "name": 1
10    },
11    "lsid": {
12       "id": {
13          "$binary": {
14             "base64": "sW+jwdW4RaucJYr6ZA9h6w==",
15             "subType": "04"
16          }
17       }
18    },
19    "$db": "restaurants"
20 }
```

Através do JSON reconstruído, é possível verificar o tipo da operação. As operações de busca de dados possuem o termo ***find*** como chave e a coleção da busca como valor correspondente. De forma semelhante, as operações de inserção, atualização e de exclusão possuem os termos *insert*, *update* e *delete*, respectivamente. Portanto, através de um simples atributo chave-valor do JSON, é possível obter o tipo da operação e em qual coleção do banco de dados ela está sendo realizada.

Porém, apenas o nome da coleção do banco de dados não é suficiente, é necessário obter o nome do banco de dados em si, já que o MongoDB possui suporte a múltiplos bancos de dados simultâneos. De forma semelhante ao tipo da operação, é possível obter diretamente o nome do banco de dados através do valor do atributo *"$db"*.

Por fim, é necessário obter a chave do dado que está sendo manipulado. Esta, por sua vez, está presente no campo *filter* do JSON e pode ser apresentada de diferentes formas, já que este atributo é montado conforme o filtro que o usuário requisitou. De acordo com a documentação do MongoDB, através do campo *filter* é possível realizar diferentes combinações, como busca por chaves iguais, maiores ou menores do que um inteiro (caso a chave seja um inteiro), por igualdade ou existência de uma *string* dentro de outra, conjunções, disjunções, entre outros. Para simplificar esta implementação, o foco foi apenas nas operações com chaves idênticas, ou seja, a requisição pela chave *"Mc"* não terá os mesmos resultados da requisição pela chave *"McDonalds"*. Portanto, nesta implementação da VNF-Cache, a chave da requisição é o valor que está no campo "$eq", que está localizado dentro do campo *"filter"*. Opcionalmente, pode-se omitir o campo "$eq", deixando a chave buscada ser diretamente o valor referente à chave daquela coleção.

Uma limitação desta implementação, que pode ser definida como um trabalho futuro, é a fixação do banco de dados e da coleção que será armazenada na cache. Esta limitação pode ser solucionada, por exemplo, ao adicionar novos recursos ao VNF-Cache *Storage*, como campos extras que armazenem qual é o banco de dados e a coleção daquele conjunto chave-valor. Portanto, esta implementação da VNF-Cache realiza a filtragem pelos pacotes que possuam os termos *find*, *insert*, *update* ou *delete* e que tenham como respectivo valor a coleção que foi fixada no módulo. Além disso, o valor do termo *"$db"* deve ser idêntico ao banco de dados, que também foi fixado no módulo.

As operações de inserção de dados não são tratadas internamente pela VNF-Cache, já que não influenciam diretamente os dados já armazenados nela. Desta forma, o *Manager* permite que os pacotes de inserção de dados enviados pelos clientes e seus respectivos pacotes de resposta fluam pela VNF-Cache de forma transparente e sem alterações. Ou seja, o VNF-Cache *Manager* não armazena o dado na cache durante o evento de inserção em si, e sim somente após a primeira requisição da chave por algum cliente. Quando isto ocorrer, se possível, a cache armazenará o valor retornado pelo servidor e, a partir da segunda requisição em diante, o dado pode ser diretamente retornado ao cliente, sem a necessidade de reencaminhar ao servidor.

4.4.3.3 Armazenamento em Dicionário Python

Ao encontrar um pacote de busca/leitura de uma chave específica, a VNF-Cache verifica primeiramente se este conjunto chave-valor já está no armazenamento local da cache. Na implementação da VNF-Cache, o módulo *Storage* de armazenamento dos dados é realizado em um dicionário Python, utilizando o mesmo par chave-valor do MongoDB. Desta forma, a chave requisitada pelo cliente é a chave do dicionário e o valor desta mesma chave no banco de dados é o respectivo valor armazenado no VNF-Cache *Storage*.

Se a chave requisitada não estiver no dicionário, o *Client Manager* reencaminha o pacote para o servidor MongoDB e aguarda pelo pacote de resposta. Se este pacote de resposta for válido e possuir o valor da chave requisitada, o *Server Manager* armazena de forma íntegra os bytes do pacote no dicionário. Por outro lado, se a chave requisitada estiver no dicionário, o *Client Manager* reconstrói o pacote de dados e o encaminha diretamente ao cliente. Desta forma, o pacote de requisição da chave enviado pelo cliente sofre um *drop*, ou seja, o pacote é ignorado e não é reencaminhado para o servidor.

A reconstrução consiste em montar um novo pacote de rede utilizando como base os mesmos bytes armazenados na cache anteriormente, com exceção dos 4 bytes de índices 8 a 12, que representam o campo ***response_to*** do pacote PyMongo. Neste caso, a VNF-Cache realiza a substituição destes bytes no novo pacote pelos mesmos bytes do campo ***request_id*** que estão presentes no pacote de requisição da chave pelo cliente. A Figura 4.5 mostra o comportamento na primeira requisição de uma chave e da segunda requisição da mesma chave, já armazenada na cache.

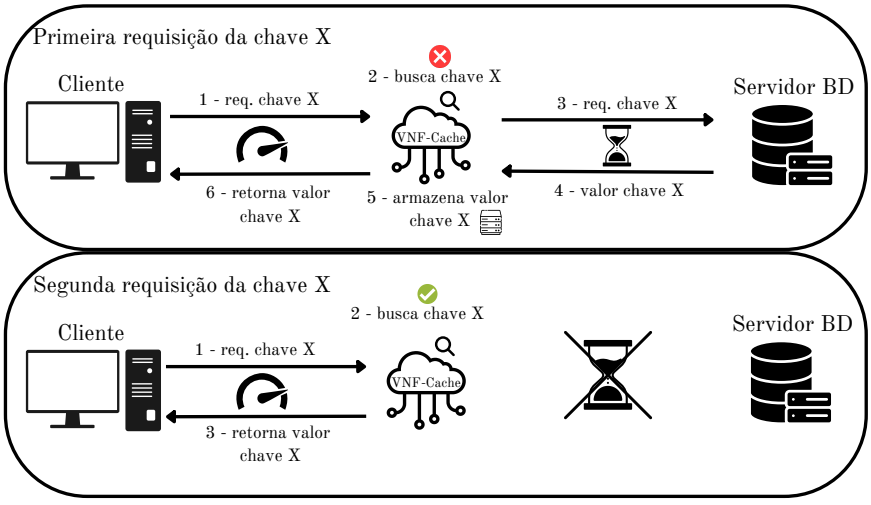


Figura 4.5: Comportamento da primeira requisição (com cache *miss*) e da segunda requisição (com cache *hit*) pela mesma chave.

4.4.3.4 Concorrência

Como existe a possibilidade de múltiplas *threads* estarem em execução ao mesmo tempo e solicitarem leituras e/ou escritas no dicionário da cache, existe a possibilidade de duas ou mais *threads* realizarem modificações no mesmo dado ao mesmo tempo, podendo causar incoerências nas respostas. Para resolver este problema, são utilizadas as primitivas ***acquire()***, que trava o acesso ao dicionário exclusivamente para aquela *thread*, e ***release()***, que libera o acesso para as demais *threads*.

4.4.3.5 Configurações e Estatísticas

De forma complementar ao funcionamento básico da VNF-Cache, nesta implementação existem opções de linha de comando que coordenam os níveis de detalhamento do *log* produzido, o número máximo de itens da VNF-Cache e a geração de arquivos estatísticos, como os de registros de cache *hit* e *miss*, por exemplo.

Para o gerenciamento do nível de detalhamento do *log*, a utilização da opção ***-debugPayload*** ativa o *logging* dos bytes de cada pacote que passou pelos fluxos da cache, que é desativado por padrão. Por sua vez, a utilização da opção ***-disablePacketInfoDebug*** desativa o *logging* do JSON dos pacotes, que é ativo por padrão. A opção ***-maxSizeCache N***, por sua vez, altera para *N* a quantidade máxima de itens que podem ser armazenados ao mesmo tempo no dicionário da cache, que é 50 por padrão. Por fim, a utilização da *flag* ***-enableStatHits*** ativa a geração do arquivo "*statHits.txt*", que escreve "0", caso a busca por uma chave seja um cache *miss* ou "1", caso a busca seja um cache *hit*.

4.4.3.6 Implementação das Políticas

Conforme apresentado na Seção 2.4, existem algumas opções de políticas de preenchimento e de substituição dos dados da cache, tendo como objetivo a melhora da eficiência e da coerência dos dados. Dentre as políticas apresentadas estão a *Write-Through*, que faz as alterações dos dados na cache e na origem simultaneamente, e a *Write-Back*, que realiza, em um primeiro momento, as alterações apenas na cache, postergando as alterações na origem para algum momento futuro.

Nesta implementação da VNF-Cache, optou-se por utilizar uma aplicação da política *Write-Invalidate*. Desta forma, as operações de alteração e exclusão dos dados fazem com que os mesmos sejam retirados do *Storage*, como uma forma de invalidar os dados. Assim, ao receber um pacote que atualiza o dado de uma chave, o respectivo conjunto chave-valor é retirado da cache e o pacote é reencaminhado para o servidor, que realiza as alterações necessárias no banco de dados. Ou seja, a partir deste momento, o primeiro pacote que solicitar este mesmo dado é encaminhado para o servidor e, se possível, seu novo valor é armazenado no *Storage* após a VNF-Cache receber o pacote de resposta. Contudo, existem algumas limitações nesta implementação, que são relatadas a seguir.

4.4.4 Limitações da Implementação

Como esta implementação da VNF-Cache é um protótipo para prova de conceito, existem algumas limitações em seu funcionamento. Embora estas limitações existam, suas soluções são possíveis e algumas estão descritas no Capítulo 5 como trabalhos futuros. Algumas limitações são descritas a seguir.

4.4.4.1 Preenchimento Sequencial e Ausência de Substituição

Na Seção 2.4.2 foram apresentadas políticas de preenchimento e substituição dos dados na cache, como a LFU, que preenche a cache com os dados mais frequentemente utilizados, e a LRU, que preenche com os dados mais recentemente utilizados. Na implementação realizada, o preenchimento dos dados no *Storage* é realizado de forma sequencial, ou seja, na ordem de recebimento das requisições pelas chaves do banco de dados. Este preenchimento segue até que o armazenamento esteja cheio e, a partir deste momento, nenhum dado novo entra na cache, e os dados já presentes na cache são substituídos apenas em caso de atualizações ou exclusões realizadas pelos clientes.

Embora esta política já apresente resultados relevantes, outras técnicas podem ser aplicadas para otimizar o uso da cache de acordo com o padrão de requisições dos clientes no contexto da implantação. Se estes clientes realizam requisições frequentes em um mesmo conjunto de chaves, a melhor opção seria armazenar na cache esse conjunto de chaves, que não foram necessariamente as primeiras requisitadas pelos clientes. Além disso, os conjuntos chave-valor armazenados de forma sequencial podem apresentar um número baixo de requisições, e a VNF-Cache pode não oferecer o desempenho esperado.

4.4.4.2 Banco de Dados e Coleção Fixos

Como relatado anteriormente, nesta implementação da VNF-Cache, o banco de dados e a coleção dos conjuntos chave-valor são fixos no código. Isto facilita no momento de realizar a filtragem dos pacotes e no armazenamento dos dados na cache. A implementação de bancos de dados e/ou coleções variáveis pode ser realizada através do uso de diferentes implementações do VNF-Cache *Storage*, como dicionários *Python* que armazenam outros dicionários *Python* como valor, por exemplo. Neste caso, a chave principal poderia ser o nome do banco de dados, e caso o banco de dados da requisição esteja em alguma entrada da cache, a verificação prosseguiria para o dicionário presente no valor desta entrada. Isto, porém, aumentaria a quantidade de processamento, potencialmente aumentando a latência e o *overhead* causado pelo módulo.

4.4.4.3 Dependência da Disponibilidade da Cache

A implementação do protótipo não é tolerante a falhas. Consequentemente, o módulo *Python* da cache precisa estar totalmente operacional na infraestrutura virtualizada, além de receber e enviar pacotes de rede para o cliente e para o servidor corretamente. Caso por algum motivo, como falta de memória ou queda de energia o módulo da cache não esteja disponível, a comunicação entre cliente e servidor será impactada.

Como solução, é possível, por exemplo, definir políticas de redirecionamento redundantes que efetuem outros redirecionamentos caso a VNF-Cache não responda aos pacotes enviados. Em um cenário com múltiplas VNF-Caches, por exemplo, o roteamento a ser realizado para a VNF-Cache *X* poderia ser redirecionado para a VNF-Cache *Y*. Desta forma, a comunicação entre cliente e servidor não é impactada e os benefícios das VNF-Caches ainda podem ser observados.

4.4.4.4 Operações com Parâmetro filter *Vazio*

Pela própria limitação da biblioteca PyMongo, nesta implementação da VNF-Cache, o uso de algumas funções como ***findOneAndReplace({})***, por exemplo, que possuam o parâmetro *filter* vazio, acabam por invalidar todos os dados do *Storage* da VNF-Cache. Isto ocorre pois estas funções, quando utilizadas sem o preenchimento de um objeto JSON como filtro, possuem comportamento pseudo-aleatório e, em algumas ocasiões, não é possível recuperar qual conjunto

chave-valor foi alterado. Essa limitação se dá ao fato de que em algumas respostas de operações de manipulação de dados, o servidor retorna apenas a quantidade de documentos alterados/excluídos, sem informar o nome do banco de dados, o nome da coleção e nem a chave que foi utilizada pelo cliente.

Desta maneira, quando ocorrerem operações com parâmetro *filter* vazio, nesta implementação da VNF-Cache, todos os registros do VNF-Cache *Storage* são excluídos. Assim, a partir deste momento, todas as requisições (independentemente da chave requisitada) precisam necessariamente serem reencaminhadas ao servidor, que retornará o valor mais recente da chave. Isto mantém a coerência dos dados entre a VNF-Cache e o servidor, evitando a propagação de dados desatualizados que poderiam invalidar o funcionamento da cache.

4.4.4.5 Comunicação Não Criptografada

Outra limitação desta VNF-Cache é a necessidade da comunicação entre o cliente e o servidor de banco de dados MongoDB através do PyMongo não ser criptografada. Embora isto represente um risco potencial para a segurança das informações trafegadas, é necessário o acesso completo aos pacotes, já que é através do processamento dos campos do cabeçalho PyMongo e do JSON enviado pelos clientes que a cache faz as decisões de redirecionamento dos pacotes.

4.4.4.6 Retorno de Múltiplos Documentos

Em um cenário no qual podem existir vários valores no banco de dados para a mesma chave, a implementação da VNF-Cache não armazena os respectivos dados no *Storage*. Ou seja, quando um servidor retornar múltiplos valores para uma chave requisitada por algum cliente, a VNF-Cache apenas redirecionará os pacotes de resposta, sem adicionar ou remover dados no armazenamento local. Isto ocorre por causa da dificuldade extra adicionada pelo gerenciamento de múltiplos valores por chave, já que é mais desafiador evitar incoerências entre as diferentes versões dos documentos no banco de dados e no armazenamento da VNF-Cache.

4.4.4.7 Concorrência de Acesso ao Armazenamento

Conforme o número de clientes aumenta, o número de conexões com o *socket* da cache também aumenta, e consequentemente o número de *threads* em execução é maior. Caso o fluxo de requisições dos clientes seja muito intenso, as *threads* podem começar a competir pelo acesso ao *Storage* para realizar as leituras ou escritas dos dados. Como a implementação restringe o acesso ao dicionário *Python* a apenas uma única *thread* por vez, as demais *threads* entram em fila e aguardam a sua liberação de acesso. Isto pode reduzir o desempenho final da VNF-Cache, podendo inclusive aumentar o tempo de resposta das requisições, causando efeito contrário ao objetivo de sua implantação.

4.4.5 Avaliação Empírica

Esta subseção apresenta uma avaliação empírica do protótipo da VNF-Cache. Foram efetuados experimentos com diferentes cenários de aplicações da VNF-Cache, variando sua capacidade de armazenamento e a sua localização em relação ao cliente e ao servidor. Através destes experimentos, são avaliados os resultados que a VNF-Cache pode oferecer, assim como os cenários que podem se beneficiar e os que podem acarretar queda de eficiência.

O método escolhido para avaliar a eficiência da implementação é o de geração de requisições para o servidor de banco de dados e a avaliação das métricas impactadas pela VNF-Cache. Dentre estas métricas, estão o tempo de resposta de uma requisição, que compreende o

intervalo entre o envio da requisição pelo cliente e a chegada da resposta enviada pelo servidor, e a quantidade de requisições processadas em um determinado intervalo de tempo. Foram definidos três cenários diferentes para os experimentos: (A) cliente, VNF-Cache e servidor de banco de dados próximos entre si, (B) cliente e VNF-Cache próximos entre si, e o servidor distante e (C) cliente, VNF-Cache e servidor distantes entre si. Desta forma, é possível analisar a eficiência das diferentes aplicações da VNF-Cache conforme a distância entre os clientes e os servidores varia.

Para a execução dos experimentos, foi utilizada uma máquina física e múltiplas combinações de máquinas virtuais, conforme será descrito adiante. A máquina física possui um processador Intel(R) Core(TM) i5-7400 @3.0 GHz x 4, 16GB de memória RAM, uma interface de rede de 100 Mb/s e sistema operacional *Ubuntu 20.04.6*. Esta máquina serviu para a coordenação dos testes e execução de algumas das máquinas virtuais. Estas, por sua vez, foram instanciadas tanto localmente utilizando *Kernel-based Virtual Machine* (KVM), quanto remotamente através da *Amazon Elastic Compute Cloud*[3], um serviço de computação em nuvem da *Amazon Web Services*[4] (AWS) que possibilita a instanciação de máquinas virtuais em diferentes localizações do mundo.

Para automatizar e facilitar a execução dos experimentos, foram implementados *scripts* utilizando *Shell* e Python. Estes realizam 30 lotes de 1000 requisições para chaves inteiras aleatórias, distribuídas de forma uniforme no intervalo de 1 a 100 da coleção *phrases* do banco de dados *randomPhrases* do MongoDB, que é constituído por frases aleatórias. Em conjunto, estes *scripts* atuam como um cliente, realizando requisições das informações diretamente para o endereço do servidor. Além disso, estes também realizam a medição do tempo de resposta de cada requisição e do número de requisições processadas por segundo.

Por sua vez, a VNF-Cache foi executada em máquinas virtuais com duas opções de especificações. As máquinas virtuais executadas na máquina física utilizam o sistema operacional *Ubuntu 20.04*, em um processador virtualizado de 3 GHz x 2, memória principal de 2 GB e 15 GB de armazenamento em disco. Já na AWS, as máquinas virtuais executam o mesmo sistema operacional, porém sobre um processador virtualizado de 2.5 GHz x 1, memória principal de 1 GB e 8 GB de armazenamento em disco. Por fim, o servidor de banco de dados MongoDB foi implementado em máquinas virtuais com *Ubuntu Server 20.04*, processador de 1 GHz (no KVM) ou de 2,5 GHz (na AWS), 1 GB de memória RAM e 10GB de armazenamento em disco.

As seguintes seções apresentam os experimentos e seus resultados para cada cenário. A Subseção 4.4.5.1 apresenta o cenário A, realizado com o cliente, VNF-Cache e servidor de banco de dados localizados próximos entre si. A Subseção 4.4.5.2 apresenta o cenário B, experimentos realizados com o cliente e a VNF-Cache próximos, distanciando geograficamente o servidor. Por fim, Subseção 4.4.5.3 apresenta os experimentos do cenário C, realizados distanciando o cliente, a VNF-Cache e o servidor entre si.

4.4.5.1 (A) Cliente, VNF-Cache e Servidor Próximos

O experimento com cliente, VNF-Cache e servidor de banco de dados próximos entre si foi o primeiro a ser executado. Para isso, foram instanciadas três máquinas virtuais na máquina física, variando as especificações de acordo com a funcionalidade, conforme citado anteriormente. Além disso, foram implementadas políticas de redirecionamento no roteador da rede criada para intercomunicação das VMs. Desta forma, todo pacote do cliente com destino ao banco de dados do servidor é desviado para uma porta específica da VM que executa a VNF-Cache. A Tabela

[3]https://aws.amazon.com/pt/ec2/?nc2=h_ql_prod_cp_ec2

[4]https://aws.amazon.com/pt/

4.1 mostra os tempos de resposta, em milissegundos (ms), das requisições para a mesma chave nos experimentos sem e com a VNF-Cache.

Tabela 4.1: Tempos das requisições (em ms) para a mesma chave e *overhead* causado pela VNF-Cache no cenário A.

	Ordem das Requisições para uma Determinada Chave						
VNF-Cache	1ª	2ª	3ª	4ª	5ª	6ª	...
S/ VNF-Cache	1,12 ms	1,56 ms	2,04 ms	1,26 ms	1,17 ms	1,35 ms	...
C/ VNF-Cache	10,54 ms	5,40 ms	3,87 ms	3,32 ms	4,05 ms	3,93 ms	...
Overhead	9,42 ms	3,84 ms	1,83 ms	2,06 ms	2,88 ms	2,58 ms	...

Primeiramente, foram executados os lotes de requisições diretamente entre o cliente e o servidor, sem utilizar a VNF-Cache. Este experimento aponta que o tempo de resposta médio para cada requisição é de 1,28 milissegundos (ms) e a média de requisições por segundo é de 535. Em seguida, foram executados os lotes de requisições com o desvio para a VNF-Cache habilitado. Neste experimento, a cache foi definida para possuir capacidade máxima de 100 conjuntos chave-valor, ou seja, todo o espaço de chaves do banco de dados. Desta vez, considerando o preenchimento da cache, o tempo de resposta médio foi de 4,38 ms, ou seja, aproximadamente 3,5 vezes mais demorado. Já o número médio de requisições por segundo foi de 191, ou seja, uma queda de cerca de 65%.

Estes resultados possibilitam concluir que em um cenário de proximidade entre cliente e servidor, a VNF-Cache não atinge o seu objetivo de redução do tempo de resposta. Neste caso, isto acontece pois a VNF-Cache apenas adiciona um processamento extra no percurso dos pacotes de rede, que ao invés de trafegarem diretamente entre cliente e servidor, precisam passar pela VNF-Cache antes de chegarem aos seus destinos finais. Como neste caso o tempo de resposta das requisições diretas entre o cliente e servidor já é baixo, o *overhead* causado pela VNF-Cache, não é capaz de justificar sua implementação neste cenário. É possível perceber pela Tabela 4.1 que na primeira requisição pela chave, quando ocorre um cache *miss* e a VNF-Cache necessita requisitar o servidor, o *overhead* causado é de cerca de 9,5 ms.

Outro experimento realizado neste cenário foi a medição do *overhead* causado pela VNF-Cache no tráfego dos pacotes de inserção de dados, que nesta implementação não são tratados diretamente e fluem livremente através da VNF-Cache. Para isso, foram executadas 30.000 requisições de inserção de dados no servidor de banco de dados. Assim como nos experimentos anteriores, estas requisições foram realizadas sem e com o desvio para a VNF-Cache. Os resultados deste experimento mostram que as requisições de inserção diretas entre cliente e servidor apresentaram um tempo de resposta médio de 0,79 ms. Já as requisições de inserção de dados com a VNF-Cache no caminho apresentaram um tempo médio de 10,74 ms. Ou seja, nos pacotes que trafegam livremente pela VNF-Cache, o *overhead* de processamento adicionado é de aproximadamente 10 ms. Este *overhead* já era esperado, visto que a implementação realizada não está otimizada com nenhuma biblioteca de tratamento de pacotes de rede e a linguagem escolhida para a construção do protótipo (Python) não possui o melhor desempenho para tal.

Portanto, mesmo ao utilizar uma cache com capacidade para todos os dados do banco de dados de origem, a implementação de uma VNF-Cache neste cenário de proximidade entre cliente e servidor pode apresentar um desempenho semelhante ou pior ao de não se utilizar. Quando uma chave requisitada for um cache *hit*, seu valor retornará diretamente para o cliente, dispensando o reencaminhamento para o servidor. Porém, como neste caso o tempo de resposta do servidor para o cliente e da cache para o cliente são semelhantes, pouca ou nenhuma redução no tempo de resposta pode ser observada. Já quando uma requisição na VNF-Cache resultar em

um cache *miss*, a latência do tempo de resposta tende a ser pior, já que é necessário reencaminhar o pacote para o servidor e aguardar pela sua resposta antes de retornar ao cliente.

Por outro lado, caso o servidor de banco de dados esteja sobrecarregado e/ou apresente um alto tempo de resposta, este cenário de proximidade ainda pode se beneficiar com a implantação de uma VNF-Cache. Desta forma, mesmo que a VNF-Cache e o servidor estejam próximos ao cliente, se o servidor estiver sobrecarregado, a VNF-Cache pode retornar os dados mais rapidamente e o tráfego para o servidor pode ser amenizado.

4.4.5.2 (B) Cliente e VNF-Cache Próximos, Servidor Distante

Em seguida, os experimentos afastando geograficamente o servidor de banco de dados do cliente e da VNF-Cache foram realizados. Para isso, foram instanciadas na AWS duas máquinas virtuais para o servidor de banco de dados, sendo a primeira delas em Ohio, na costa leste dos Estados Unidos, e a segunda em Tóquio, capital do Japão. O cliente e a VNF-Cache foram executados em máquinas virtuais no KVM da máquina física em Curitiba. Ambos os bancos de dados foram populados com a mesma coleção utilizada nos testes anteriores. A Tabela 4.2 mostra os tempos de acesso para cada combinação de localizações e capacidades da VNF-Cache. Para uma melhor análise do desempenho da implementação, nas execuções deste cenário variou-se também a capacidade da VNF-Cache entre 10, 30, 70 e 100 conjuntos chave-valor.

De forma semelhante ao cenário A, primeiramente foram executados os lotes sem a intervenção da VNF-Cache, seguidos pelos lotes com o desvio para a mesma. Nos experimentos com o servidor em Ohio, o tempo médio das requisições diretas entre cliente e servidor foi de cerca de 164 ms, com cerca de 6 requisições processadas por segundo. Já nos experimentos com o servidor em Tóquio, o tempo médio das requisições diretas entre cliente e servidor foi de cerca de 292 ms, com cerca de 3,3 requisições processadas por segundo. Neste cenário, os experimentos com o desvio dos pacotes do cliente para a VNF-Cache apresentaram resultados muito satisfatórios.

Tabela 4.2: Tempos médios das requisições (em ms) para cada capacidade da VNF-Cache local e posicionamento do servidor de banco de dados chave-valor.

	Sem	Capacidade da VNF-Cache em conjuntos chave-valor			
Localização	VNF-Cache	10	30	70	100
OHIO	164 ms	174,48 ms	138,51 ms	64,66 ms	8,08 ms
JAPÃO	292 ms	303,34 ms	239,35 ms	112,66 ms	11,02 ms

Tomando como exemplo a VNF-Cache definida para ter a capacidade de 100 conjuntos chave-valor, ou seja, o mesmo tamanho do espaço de chaves do banco de dados, as requisições com o cliente e a VNF-Cache em Curitiba e o servidor de banco de dados em Ohio apresentaram um tempo médio de resposta de 8 ms com uma média de 118 requisições processadas por segundo. Já com o servidor em Tóquio e a VNF-Cache com a mesma capacidade de 100 conjuntos chave-valor, os resultados são ainda mais expressivos (se comparados aos resultados das requisições diretas entre o cliente em Curitiba e o banco de dados em Tóquio). Nestes experimentos, a implantação da VNF-Cache reduziu o tempo das requisições para uma média de 11 ms e aumentou o número de requisições processadas por segundo para uma média de 87.

Desta forma, os objetivos principais de redução do tempo de acesso aos dados e aumento do número de requisições processadas foram alcançados, já que foi obtida uma melhora de até aproximadamente 95% com o servidor em Ohio e 96% com o servidor em Tóquio. Ë evidente que, na medida em que a distância entre os clientes e os servidores aumenta, os benefícios de se utilizar a VNF-Cache também aumentam.

Um ponto relevante observado através da Tabela 4.2 é a relação de desempenho entre a capacidade da VNF-Cache local em comparação ao espaço de chaves possíveis do banco de dados. Por exemplo, nos experimentos realizados com a cache com capacidade de apenas 10 conjuntos chave-valor, é observado um pequeno aumento no tempo médio de resposta das requisições. Com o servidor em Tóquio e a VNF-Cache de 10 posições na mesma localização do cliente, o tempo médio chegou a piorar em cerca de 3,5%. Já com o servidor localizado em Ohio e a VNF-Cache com a mesma capacidade, o tempo médio de resposta das requisições piorou em cerca de 6%. Ou seja, nestes experimentos da VNF-Cache, caso a capacidade da cache seja muito pequena, o tempo de resposta médio tende a ser pior se comparado ao tempo das requisições diretas. Porém, uma VNF-Cache de baixa capacidade que implemente uma boa política de inclusão e substituição dos dados, ainda pode apresentar uma melhora de desempenho nas requisições, já que os dados mais frequentemente requisitados estarão na cache e o número de reencaminhamentos para o servidor tende a diminuir.

A Figura 4.6 mostra o comportamento das requisições por dados do servidor localizado em Ohio quando a consulta na VNF-Cache resulta em cache *miss* e quando resulta em cache *hit*. As barras de cor laranja mostram os tempos de resposta para as 10 primeiras requisições por uma chave que não foi armazenada na cache. Neste caso, as requisições apresentaram uma média de 180 ms. As barras verdes, por sua vez, mostram as 10 primeiras requisições por uma chave que foi armazenada na VNF-Cache. Desta vez, a primeira requisição apresentou um tempo de resposta de quase 200 ms e, a partir da segunda requisição, quando o valor da chave já foi armazenado na cache, o tempo médio das requisições cai para uma média de 6,5 ms. Ou seja, é possível perceber o benefício proporcionado pela VNF-Cache, já que neste caso o tempo de requisição reduziu em cerca de 96%. Além disso, por decorrência da redução no número de requisições para o servidor, o tráfego de pacotes na rede e a sobrecarga dos servidores também é reduzida.

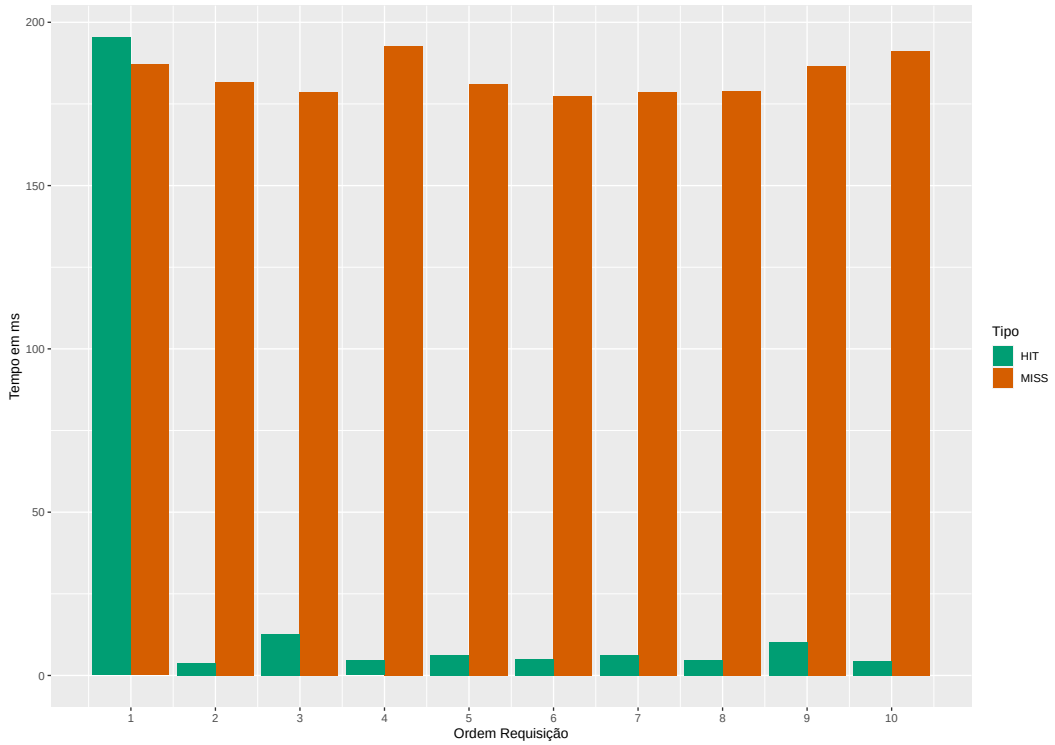


Figura 4.6: Tempos das requisições por chaves quando a consulta na VNF-Cache resulta em cache *miss* ou cache *hit*.

Por sua vez, a Figura 4.7 mostra a densidade dos tempos de resposta das requisições para o servidor em Tóquio conforme a capacidade da VNF-Cache varia. O gráfico da esquerda mostra a densidade das requisições que resultaram em cache *hit*. Neste caso, a grande maioria das requisições com a VNF-Cache com capacidade de 100 conjuntos chave-valor estão concentradas entre 5 e 15 ms. Conforme a capacidade da VNF-Cache vai diminuindo, a concentração das requisições no gráfico de *hits* também diminui, já que mais requisições tendem a ter cache *miss* e necessitam aguardar pela resposta do servidor. Outro ponto importante observado no gráfico de cache *hits* é a rara presença de requisições com a VNF-Cache com capacidade de 10 posições. Neste caso, a grande maioria dos conjuntos chave-valor requisitados não puderam ser armazenados na cache, e o tempo de resposta é maior do que a escala do gráfico.

Já no gráfico de cache *miss* da direita, a maioria das requisições da VNF-Cache com 10 conjuntos chave-valor se acumulam com tempo de resposta entre 300 e 350 ms. Neste caso, conforme a capacidade da VNF-Cache aumenta, a densidade de requisições no gráfico diminui, já que ocorrem mais cache *hits* e o tempo de resposta diminui. Desta vez, é notável a rara presença de requisições de cache *miss* com a VNF-Cache com capacidade de 100 conjuntos chave-valor. Neste caso, isto ocorre pois apenas as primeiras requisições pelas chaves resultam em cache *miss* e, consequentemente, apresentam tempo de resposta mais longo. Com a cache com capacidade de 100 conjuntos, a grande maioria das requisições resultam em cache *hit*, os tempos de resposta são menores e a densidade de requisições no gráfico de cache *miss* é reduzida. Em suma, quanto maior a capacidade da VNF-Cache, maior é a densidade das requisições que apresentam tempo de resposta menor do que o tempo das requisições diretas entre cliente e servidor. Em contrapartida, quanto menor a capacidade a VNF-Cache, maior é a densidade das requisições com tempo de resposta maior do que o tempo das requisições diretas.

4.4.5.3 (C) Cliente, VNF-Cache e Servidor Distantes

Como uma variação do cenário B, experimentos afastando a VNF-Cache do cliente também foram realizados. Para isto, através da instanciação de uma máquina virtual na AWS, os experimentos foram repetidos com a VNF-Cache localizada em São Paulo, ou seja, mais distante do cliente localizado em Curitiba. Através destes experimentos é possível analisar a eficiência da utilização de uma VNF-Cache em um cenário, no qual a cache não está localizada diretamente na mesma localização e/ou máquina do cliente. A Tabela 4.3 mostra o tempo médio das requisições apresentadas por estes experimentos.

Tabela 4.3: Tempos médios das requisições (em ms) para cada capacidade da VNF-Cache em SP e posicionamento do servidor de banco de dados chave-valor.

	Sem	Capacidade da VNF-Cache em conjuntos chave-valor			
Localização	VNF-Cache	10	30	70	100
OHIO	164 ms	134,56 ms	108,92 ms	57,88 ms	21,13 ms
JAPÃO	292 ms	257,08 ms	204,31 ms	100,50 ms	22,13 ms

De forma semelhante aos experimentos do cenário B, os experimentos do cenário C também apresentaram resultados muito satisfatórios. Com a VNF-Cache com capacidade de 100 conjuntos chave-valor, o tempo médio das requisições diminuiu em cerca de 87% com o servidor em Ohio e em cerca de 92% com o servidor em Tóquio. Já o número de requisições aumentou para uma média de 45 requisições por segundo com o servidor em Ohio (contra as 6 requisições por segundo nas requisições diretas) e para uma média de 40 requisições por segundo com o servidor em Tóquio (contra as 3,3 requisições por segundo nas requisições diretas).

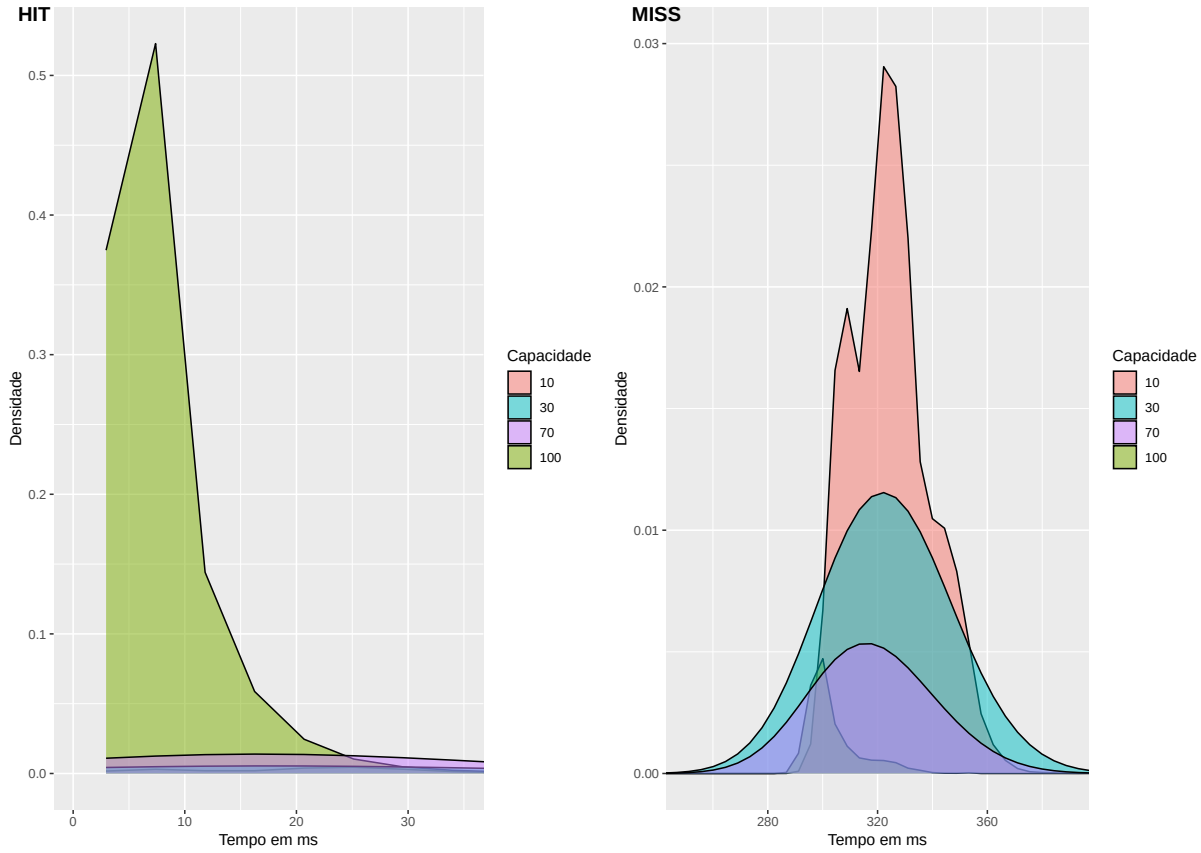


Figura 4.7: Densidade dos tempos das requisições por capacidade da VNF-Cache quando ocorre cache *hit* ou cache *miss*. Observar que as escalas de tempo nos gráficos são distintas.

Outro ponto relevante apresentado na Tabela 4.3 é o fato de a VNF-Cache já apresentar um resultado benéfico mesmo com a baixa capacidade de 10 conjuntos chave-valor. Neste caso, o tempo de resposta das requisições reduziu em cerca de 17% com o servidor em Ohio e cerca de 12% com o servidor em Tóquio. Esse resultado se opõe ao apresentado no cenário A, no qual a VNF-Cache com capacidade de apenas 10 conjuntos chave-valor apresentou um acréscimo no tempo médio de resposta das requisições. Os principais causadores deste acréscimo no tempo de resposta do cenário A (se comparado com a redução apresentada nos experimentos do cenário B) são as limitações da máquina física que executa as máquinas virtuais utilizando a tecnologia de virtualização KVM. Outro ponto que pode ter beneficiado o menor tempo de resposta do cenário C é o fato das máquinas virtuais instanciadas na AWS possuirem menos carga de trabalho do que a máquina física. Desta forma, estas máquinas virtuais possuem mais recursos computacionais para poder realizar o processamento dos pacotes de rede.

A Figura 4.8 mostra a densidade dos tempos das requisições para o servidor em Ohio conforme a capacidade da VNF-Cache localizada em São Paulo varia. Assim como no cenário B, as requisições com a VNF-Cache definida com capacidade de 100 conjuntos chave-valor se concentram no gráfico de cache *hit* e as requisições com a VNF-Cache com capacidade de 10 conjuntos chave-valor se concentram no gráfico de cache *miss*.

Por fim, a Figura 4.9 mostra a quantidade de requisições processadas por segundo durante os primeiros 60 segundos da execução dos lotes de requisições com a VNF-Cache em São Paulo e o servidor em Ohio. Esta figura aponta que durante o experimento das requisições diretas entre o cliente e o servidor, a média de requisições processadas por segundo se manteve relativamente estável, variando entre 5 e 6. Já com a VNF-Cache em São Paulo definida para

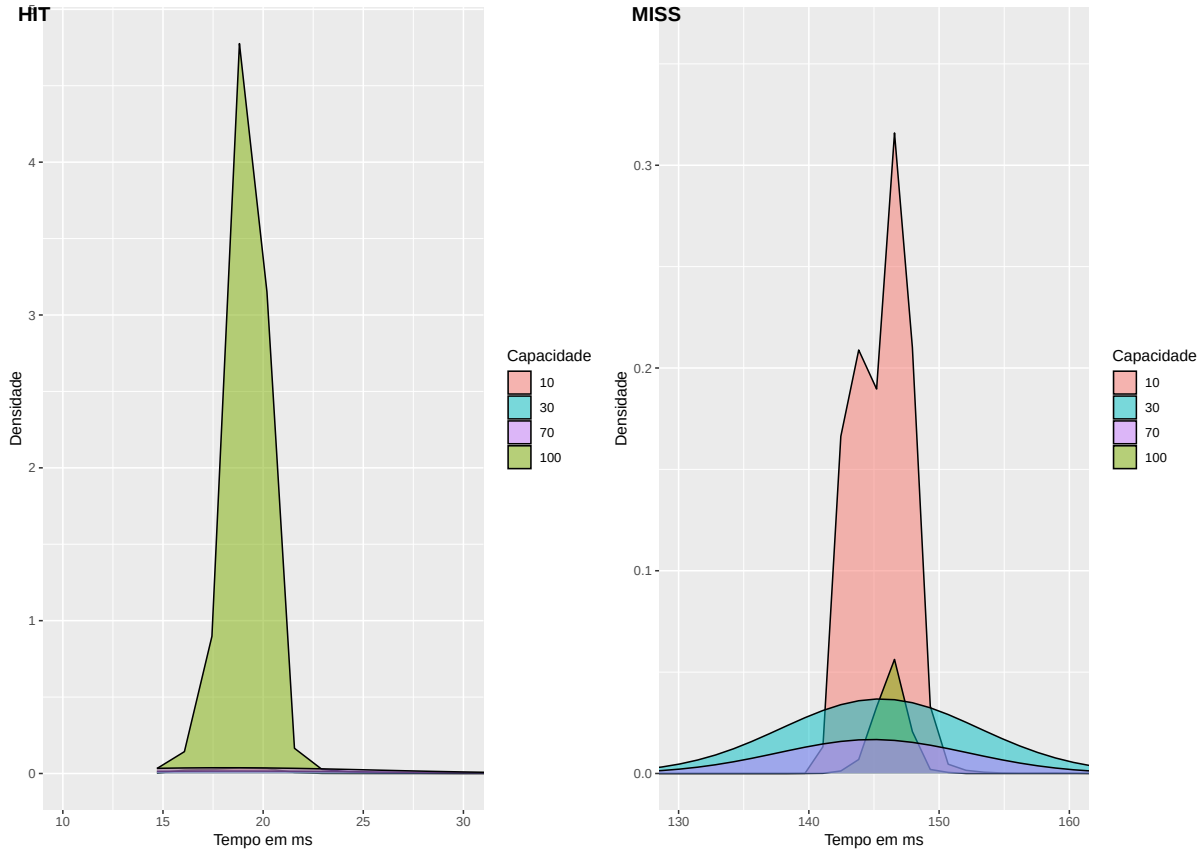


Figura 4.8: Densidade dos tempos das requisições por capacidade da VNF-Cache quando ocorre cache *hit* ou cache *miss* (cenário C).

ter a capacidade de 100 conjuntos chave-valor, após algum tempo de envio das requisições dos lotes, a média aumentou consideravelmente, ultrapassando em alguns momentos a marca de 50 requisições processadas por segundo. Este tempo no qual a média de requisições processadas por segundo ainda é baixa pode ser associado ao tempo de *warm-up* da VNF-Cache, ou seja, o tempo que demora para ela armazenar localmente as cópias dos valores das chaves. Desta forma, quando a cache está cheia, mais valores poderão ser retornados rapidamente para o cliente, aumentando o número de requisições processadas por segundo. Além disso, através da figura é possível perceber também que conforme a capacidade da VNF-Cache aumenta, maior é a média da quantidade de requisições processadas por segundo.

Através destes experimentos é comprovado que o uso da VNF-Cache pode proporcionar a redução do tempo de resposta das requisições para bancos de dados chave-valor. Além disso, o aumento do número de requisições processadas por segundo, a redução do tráfego de rede e da sobrecarga computacional também são outros pontos relevantes. Por fim, é notável a flexibilidade de implantação proporcionada pela VNF-Cache, já que a mesma pode ser instanciada em diversos pontos da rede de maneira simples e rápida.

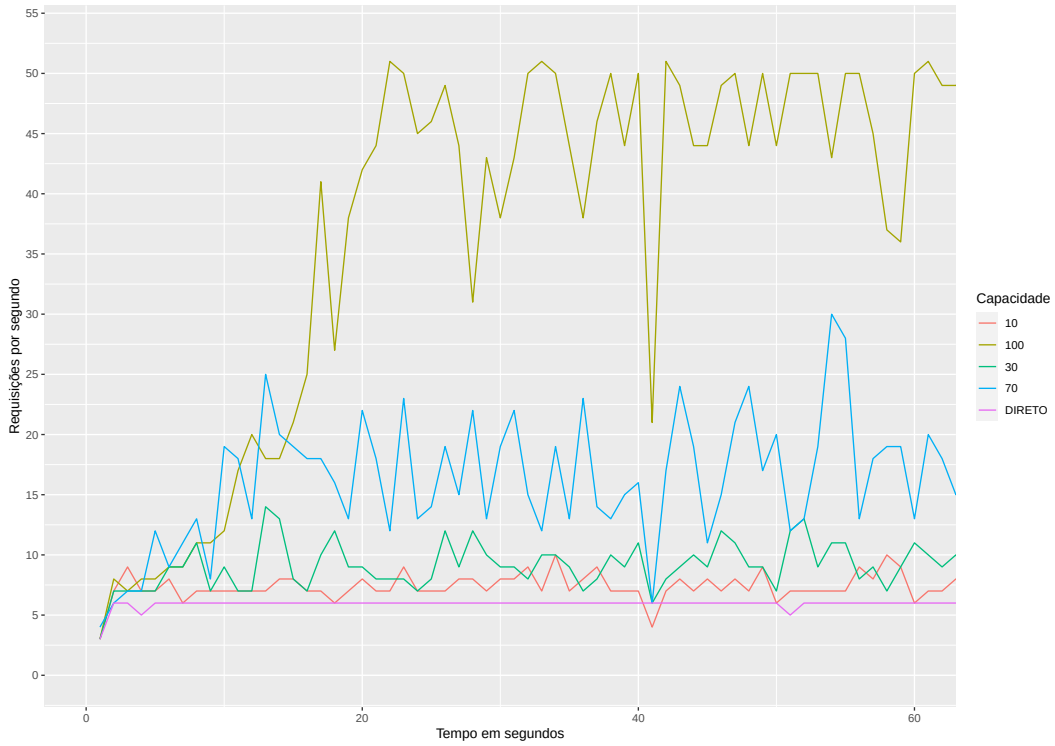


Figura 4.9: Requisições por segundo durante os primeiros 60 segundos dos experimentos para cada capacidade da VNF-Cache.

5 CONCLUSÃO

Este trabalho propôs a VNF-Cache, um serviço de cache para bancos de dados chave-valor baseado em funções de rede virtualizadas. Através do processamento dos pacotes de rede enviados entre clientes e servidores de bancos de dados, a VNF-Cache pode realizar o armazenamento de conjuntos chave-valor diretamente na rede, aproximando os dados das aplicações solicitantes. Ao retornar os dados requisitados diretamente para os clientes, a VNF-Cache possibilita uma redução no tempo de resposta, do tráfego de dados e do uso de recursos computacionais. Herdando as características de NFV, a VNF-Cache possui ainda uma maior flexibilidade nas aplicações, podendo ser adaptada conforme o contexto em que está sendo utilizada.

Através da implementação de um protótipo e dos experimentos realizados, foi possível obter uma redução considerável no tempo de resposta das requisições para os servidores de bancos de dados chave-valor distantes. Além disto, os experimentos apontam também um aumento expressivo do número de requisições processadas por segundo. Outro ponto de destaque é o aumento do desempenho da VNF-Cache conforme a distância entre o cliente e o servidor aumenta. Isto ocorre pois quanto mais distante o servidor está, maior tende a ser o tempo de resposta das requisições diretas. Nestes casos, a utilização de uma VNF-Cache próxima aos clientes tende a diminuir o tempo de resposta das requisições para valores compatíveis com a distância entre o cliente e a VNF-Cache, independentemente da distância entre a VNF-Cache propriamente dita e o servidor de banco de dados remoto.

5.1 TRABALHOS FUTUROS

As atuais limitações deste trabalho podem ser utilizadas como motivação para o desenvolvimento de trabalhos futuros. A principal limitação da implementação da VNF-Cache atualmente é a política simples de preenchimento e substituição de dados. Utilizar políticas como as descritas na Seção 2.4.2 em trabalhos futuros pode otimizar ainda mais o acesso aos dados e a quantidade de armazenamento utilizada pela VNF-Cache. Outra alteração que ampliaria ainda mais o escopo de funcionamento da proposta é possibilidade de *caching* de dados de bancos de dados e coleções variáveis. Além disso, possibilitar o uso de múltiplos bancos de dados chave-valor simultaneamente, como o Redis e o Amazon DynamoDB, expandiria ainda mais a relevância da proposta.

Outra limitação da implementação que poderia ser solucionada em um trabalho futuro é a necessidade de comunicação segura entre os clientes e os servidores. No momento atual da segurança computacional, utilizar métodos de comunicação não criptografados na Internet representa um risco para a proteção de dados sensíveis. Neste sentido, são necessários mais estudos para viabilizar o uso de VNF-Caches com segurança. Por fim, outro trabalho futuro que pode ser relevante é a implementação de métodos de armazenamento dos conjuntos chave-valor em estruturas de dados mais robustas e que possuam um melhor tratamento para manipulações de dados concorrentes. A atual implementação da VNF-Cache possibilita apenas um acesso por vez aos dados da cache, fato que pode ocasionar gargalos se múltiplos clientes estiverem requisitando e/ou manipulando as mesmas informações.

REFERÊNCIAS

Amiri, K., Park, S., Tewari, R. e Padmanabhan, S. (2003). Dbproxy: a dynamic data cache for web applications. Em *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, páginas 821–831.

Bondan, L., Franco, M. F., Marcuzzo, L., Venancio, G., Santos, R. L., Pfitscher, R. J., Scheid, E. J., Stiller, B., De Turck, F., Duarte, E. P. et al. (2019). Fende: marketplace-based distribution, execution, and life cycle management of vnfs. *IEEE Communications Magazine*, 57(1):13–19.

Cao, P. e Irani, S. (1997). Cost-Aware WWW proxy caching algorithms. Em *USENIX Symposium on Internet Technologies and Systems (USITS 97)*, Monterey, CA. USENIX Association.

Clayman, S., Kalan, R. S. e Sayit, M. (2018). Virtualized cache placement in an sdn/nfv assisted sand architecture. Em *2018 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, páginas 1–5. IEEE.

Einziger, G., Eytan, O., Friedman, R. e Manes, B. (2018). Adaptive software cache management. Em *Proceedings of the 19th International Middleware Conference*, Middleware '18, página 94–106, New York, NY, USA. Association for Computing Machinery.

ETSI (2012). Network functions virtualisation – introductory white paper. Standard, European Telecommunications Standards Institute, Darmstadt, Germany.

ETSI (2021). Etsi gr nfv-man 001 v1.2.1 - network functions virtualisation (nfv); management and orchestration; report on management and orchestration framework. Standard, European Telecommunications Standards Institute, Valbonne, France.

Flauzino, J. e Duarte Jr, E. P. (2022). Uma plataforma nfv-mano para suporte e orquestraçao de serviços de rede virtualizados em nuvem cloudstack. Em *Anais Estendidos do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, páginas 169–176. SBC.

Flauzino, J. e Jr., E. D. (2022). Uma plataforma nfv-mano para suporte e orquestração de serviços de rede virtualizados em nuvem cloudstack. Em *Anais Estendidos do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, páginas 169–176, Porto Alegre, RS, Brasil. SBC.

Fulber-Garcia, V., Duarte Jr, E. P., Huff, A. e dos Santos, C. R. (2020a). Network service topology: Formalization, taxonomy and the custom specification model. *Computer Networks*, 178:107337.

Fulber-Garcia, V., Luizelli, M. C., dos Santos, C. R. P. e Duarte, E. P. (2020b). Cusco: a customizable solution for nfv composition. Em *Advanced Information Networking and Applications: Proceedings of the 34th International Conference on Advanced Information Networking and Applications (AINA-2020)*, páginas 204–216. Springer.

Garcia, V. F., Marcuzzo, L. d. C., Huff, A., Bondan, L., Nobre, J. C., Schaeffer-Filho, A., dos Santos, C. R., Granville, L. Z. e Duarte, E. P. (2019). On the design of a flexible architecture for virtualized network function platforms. Em *2019 IEEE Global Communications Conference (GLOBECOM)*, páginas 1–6. IEEE.

Handy, J. (1993). *The Cache Memory Book*. Academic Press Professional, Inc., USA.

Huff, A., Venâncio, G., Garcia, V. F. e Duarte, E. P. (2020). Building multi-domain service function chains based on multiple nfv orchestrators. Em *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, páginas 19–24. IEEE.

Idreos, S. e Callaghan, M. (2020). Key-value storage engines. Em *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, página 2667–2672, New York, NY, USA. Association for Computing Machinery.

Jacob, B., Ng, S. e Wang, D. (2008). *Memory Systems: Cache, DRAM, Disk*. Elsevier Inc.

Juels, A., Jakobsson, M. e Jagatic, T. (2006). Cache cookies for browser authentication. Em *2006 IEEE Symposium on Security and Privacy (S&P'06)*, páginas 5 pp.–305.

Liu, Y., Point, J. C., Katsaros, K. V., Glykantzis, V., Siddiqui, M. S. e Escalona, E. (2017). Sdn/nfv based caching solution for future mobile network (5g). Em *2017 European Conference on Networks and Communications (EuCNC)*, páginas 1–5. IEEE.

Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R. e Huici, F. (2014). Clickos and the art of network function virtualization. Em *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, página 459–473, USA. USENIX Association.

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

Patterson, D. A. e Hennessy, J. L. (2014). *The Basics of Caches*, página 372–498. Morgan Kaufmann is an imprint of Elsevier, 5th edition.

Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M. e Kalnis, P. (2017). In-network computation is a dumb idea whose time has come. Em *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, páginas 150–156.

Seeger, M. (2009). Key-value stores: a practical overview. *Medieninformatik*.

Shih, M.-W., Kumar, M., Kim, T. e Gavrilovska, A. (2016). S-nfv: Securing nfv states by using sgx. Em *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, páginas 45–48.

Smith, A. J. (1982). Cache memories. *ACM Comput. Surv.*, 14(3):473–530.

Sysel, M. e Doležal, O. (2014). An educational http proxy server. *Procedia Engineering*, 69:128–132.

Tavares, T. N., da Cruz Marcuzzo, L., Garcia, V. F., de Souza, G. V., Franco, M. F., Bondan, L., De Turck, F., Granville, L. Z., Junior, E. P. D., dos Santos, C. R. P. et al. (2018). Niep: Nfv infrastructure emulation platform. Em *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, páginas 173–180. IEEE.

Turchetti, R. C. e Duarte, E. P. (2015). Implementation of failure detector based on network function virtualization. Em *2015 IEEE International Conference on Dependable Systems and Networks Workshops*, páginas 19–25. IEEE.

Turchetti, R. C. e Duarte Jr, E. P. (2017). Nfv-fd: Implementation of a failure detector using network virtualization technology. *International Journal of Network Management*, 27(6):e1988.

Veitch, P., Curley, E. e Kantecki, T. (2017). Performance evaluation of cache allocation technology for nfv noisy neighbor mitigation. Em *2017 IEEE Conference on Network Softwarization (NetSoft)*, páginas 1–5. IEEE.

Venâncio, G. e Duarte Jr, E. P. (2022). Nham: An nfv high availability architecture for building fault-tolerant stateful virtual functions and services. Em *Proceedings of the 11th Latin-American Symposium on Dependable Computing*, páginas 35–44.

Venâncio, G., Garcia, V. F., da Cruz Marcuzzo, L., Tavares, T. N., Franco, M. F., Bondan, L., Schaeffer-Filho, A. E., Paula dos Santos, C. R., Granville, L. Z. e P. Duarte Jr, E. (2021a). Beyond vnfm: Filling the gaps of the etsi vnf manager to fully support vnf life cycle operations. *International Journal of Network Management*, 31(5):e2068.

Venâncio, G., Turchetti, R. C., Camargo, E. T. e Duarte Jr, E. P. (2021b). Vnf-consensus: A virtual network function for maintaining a consistent distributed software-defined network control plane. *International Journal of Network Management*, 31(3):e2124.

Venâncio, G., Turchetti, R. C. e Duarte, E. P. (2019). Nfv-rbcast: Enabling the network to offer reliable and ordered broadcast services. Em *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, páginas 1–10. IEEE.

Venâncio, G., Turchetti, R. C. e Duarte Jr, E. P. (2022). Nfv-coin: Unleashing the power of in-network computing with virtualization technologies. *Journal of Internet Services and Applications*, 13(1):46–53.

Zheng, G., Tsiopoulos, A. e Friderikos, V. (2018). Optimal vnf chains management for proactive caching. *IEEE Transactions on Wireless Communications*, 17(10):6735–6748.

Zhuang, W., Ye, Q., Lyu, F., Cheng, N. e Ren, J. (2019). Sdn/nfv-empowered future iov with enhanced communication, computing, and caching. *Proceedings of the IEEE*, 108(2):274–291.