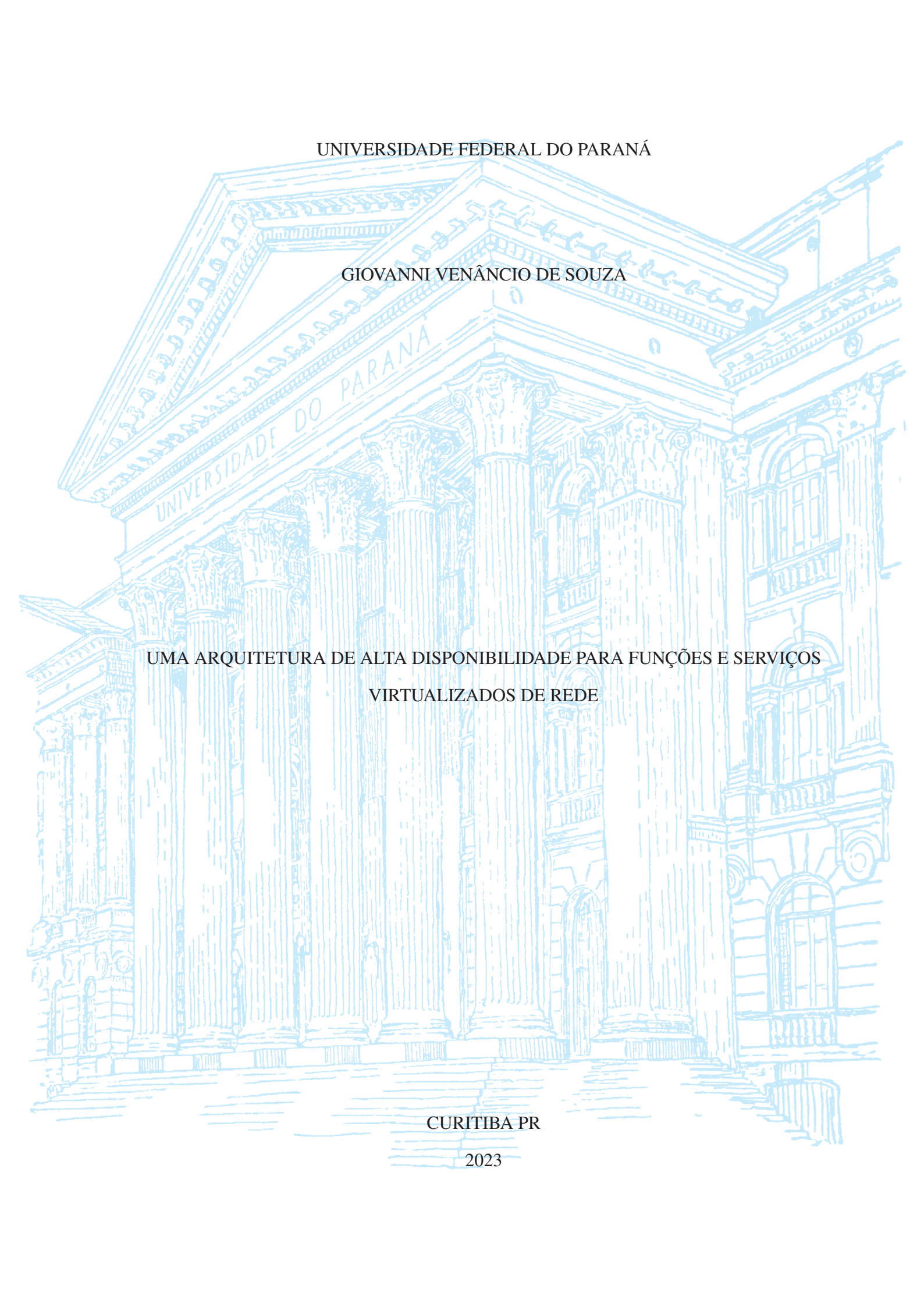




UNIVERSIDADE FEDERAL DO PARANÁ

GIOVANNI VENÂNCIO DE SOUZA

UMA ARQUITETURA DE ALTA DISPONIBILIDADE PARA FUNÇÕES E SERVIÇOS
VIRTUALIZADOS DE REDE

CURITIBA PR

2023

GIOVANNI VENÂNCIO DE SOUZA

UMA ARQUITETURA DE ALTA DISPONIBILIDADE PARA FUNÇÕES E SERVIÇOS
VIRTUALIZADOS DE REDE

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Elias P. Duarte Jr.

CURITIBA PR

2023

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA CIÊNCIA E TECNOLOGIA

Souza, Giovanni Venâncio de.

Uma arquitetura de alta disponibilidade para funções e serviços virtualizados de rede. / Giovanni Venâncio de Souza. – Curitiba, 2023.

1 recurso on-line : PDF.

Tese (Doutorado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Prof. Dr. Elias P. Duarte Jr.

1. Informática. 2. Arquitetura de redes de computador. 3. Internet. 4. Serviços na web. I. Duarte Jr., Elias P. II. Universidade Federal do Paraná. Programa de Pós-Graduação em Informática. III. Título.

Bibliotecário: Nilson Carlos Vieira Júnior CRB-9/1797

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **GIOVANNI VENÂNCIO DE SOUZA** intitulada: **Uma Arquitetura de Alta Disponibilidade Para Funções e Serviços Virtualizados de Rede**, sob orientação do Prof. Dr. ELIAS PROCÓPIO DUARTE JÚNIOR, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 10 de Fevereiro de 2023.

Assinatura Eletrônica

15/02/2023 12:00:34.0

ELIAS PROCÓPIO DUARTE JÚNIOR
Presidente da Banca Examinadora

Assinatura Eletrônica

15/02/2023 14:37:54.0

EDUARDO CUNHA DE ALMEIDA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica

26/02/2023 06:20:43.0

LUCIANO PASCHOAL GASPARY
Avaliador Externo (UNIVERSIDADE FEDERAL DO RIO GRANDE DO
SUL)

Assinatura Eletrônica

15/02/2023 14:37:02.0

EDUARDO ADILIO PELINSON ALCHIERI
Avaliador Externo (UNIVERSIDADE DE BRASÍLIA)

*Dedico este trabalho aos meus
pais Celso e Silvana e à minha
esposa Natissa.*

AGRADECIMENTOS

Em primeiro lugar, agradeço o meu orientador Prof. Elias. Foi gratificante trabalhar com você nesses últimos anos. Obrigado pelo apoio e por sempre confiar no meu trabalho.

Aos meus pais Celso e Silvana, que sempre me apoiaram e incentivaram, muito obrigado!

À minha esposa Natissa, agradeço por estar comigo durante todos estes anos, sempre me apoiando nos momentos de dificuldade. Sei que posso contar com você em todos os momentos.

Agradeço também a todas as pessoas que de alguma forma contribuíram para a realização deste trabalho.

RESUMO

A virtualização vem revolucionando a forma como as redes são construídas e gerenciadas, permitindo a sua evolução para múltiplas direções. A Virtualização de Funções de Rede (*Network Function Virtualization* - NFV) pode gerar mudanças significativas na rede, uma vez que funções de rede tradicionalmente implementadas em hardware dedicado podem ser substituídas por software, denominadas de Funções Virtualizadas de Rede (*Virtualized Network Functions* - VNFs). Apesar das vantagens, alguns desafios ainda devem ser explorados para permitir a sua ampla adoção. As contribuições propostas nesta Tese de Doutorado são divididas em três partes. A primeira parte explora o fato de que serviços virtualizados possuem maior susceptibilidade a falhas do que as suas alternativas em hardware dedicado. Tendo em vista que as redes se tornaram extremamente necessárias, é essencial garantir a execução correta e contínua dos serviços. A primeira contribuição propõe uma arquitetura NFV de alta disponibilidade para serviços de rede. A arquitetura realiza o gerenciamento de falhas, oferecendo múltiplas estratégias de recuperação, além de preservar o estado das VNFs através de técnicas de *Checkpoint/Restore*. Um protótipo da arquitetura foi implementado e resultados experimentais mostram que é possível atingir níveis de disponibilidade similares aos de sistemas comerciais de telecomunicações. A segunda parte desta Tese analisa a susceptibilidade a falhas de serviços sob outra perspectiva. Considerando que tais serviços são geralmente executados sobre uma arquitetura subjacente, a falha em algum componente desta arquitetura pode afetar toda a infraestrutura, impedindo o acesso ou possibilitando o uso não autorizado do sistema. A segunda contribuição propõe a FIT-SFC (*Fault- & Intrusion- Tolerant SFC*): uma arquitetura para suportar serviços virtuais seguros e altamente disponíveis. A FIT-SFC é baseada em técnicas de replicação para tolerar falhas por parada, por omissão ou intrusão em qualquer de seus componentes. Um protótipo da arquitetura foi implementado e resultados são apresentados para os custos para tolerar as falhas. Por fim, a terceira parte desta Tese investiga a possibilidade de utilizar NFV para permitir a execução de serviços virtualizados dentro da própria rede. No conceito chamado COIN (*COmputing In the Network*), aplicações normalmente executadas pelos próprios usuários finais passam a ser inteiramente executados nativamente dentro da rede. A terceira contribuição explora a sinergia entre NFV e COIN, denominada de NFV-COIN. Uma arquitetura é proposta para permitir a oferta e gerenciamento de serviços NFV-COIN, além de oferecer uma interface alto nível que permite a manipulação padronizada dos serviços de rede. Experimentos executados em um protótipo implementado da arquitetura demonstram a possibilidade de oferecer e executar serviços NFV-COIN sem introduzir perdas significativas de desempenho.

Palavras-chave: Virtualização de Funções de Rede. Alta Disponibilidade. Gerenciamento de Falhas. Computação Dentro da Rede.

ABSTRACT

Virtualization has represented a true revolution in the way networks are built and managed, allowing them to evolve along multiple directions. In particular, Network Function Virtualization (NFV) has been causing deep changes, since network functions traditionally implemented as specialized hardware can be replaced by software, called Virtualized Network Functions (VNFs). Despite their many advantages, some challenges still need to be addressed in order to allow its wide adoption. The contributions proposed in this Doctoral Thesis are divided into three parts. The first part explores the fact that virtualized services are more prone to failures than traditional alternatives available as specialized hardware. As networks have become extremely necessary, it is essential to ensure the correct and continuous operation of services. The first contribution proposes a high availability architecture for NFV-based services. The architecture performs fault management, offers multiple recovery strategies, while also preserving the state of VNFs through Checkpoint/Restore techniques. A prototype was implemented and experimental results show that it is possible to reach carrier-grade availability. The second part of this Thesis analyzes the susceptibility of service failures from another perspective. Considering that virtualized services usually run on an underlying architecture, the failure of any component of this architecture could affect the entire infrastructure, restraining access or allowing unauthorized use of the system. Therefore, the second contribution proposes the FIT-SFC (Fault- & Intrusion- Tolerant SFC): an architecture to support secure and highly available virtual services. The FIT-SFC architecture is based on replication techniques to tolerate crash, omission, or intrusion failures in any of its components. A prototype was implemented and results are presented for the costs to tolerate failures. Finally, the third part of this Thesis investigates the possibility of using NFV to allow the execution of virtualized services within the network. In the context of COIN (COmputing In the Network), applications that are usually executed by the end users themselves, can now be entirely executed natively within the network. The third contribution explores the synergy between NFV and COIN, called NFV-COIN. An architecture is proposed to allow the deployment and management of NFV-COIN services, while also offering a high-level interface that allows standardized operation of network services. Experiments were executed on an implemented prototype and results demonstrate the possibility of deploying and executing NFV-COIN services without introducing significant performance losses.

Keywords: Network Function Virtualization. High Availability. Network Fault Management. Computing In the Network.

LISTA DE FIGURAS

2.1	Comparação entre uma rede tradicional e uma rede baseada em NFV..	19
2.2	Arquitetura NFV alto nível proposta pela ETSI.	19
2.3	Componentes do NFV-MANO..	20
2.4	Arquitetura interna de uma SFC.	22
3.1	Execução do algoritmo PBFT.	29
4.1	Arquitetura NFV-MANO e o NHAM.	35
4.2	A arquitetura NHAM..	36
4.3	Funcionamento do mecanismo 0R.	39
4.4	Funcionamento do mecanismo 1R-AS.	39
4.5	Funcionamento do mecanismo 1R-AA..	40
4.6	Funcionamento do mecanismo MR-AA.	40
4.7	A arquitetura para SFCs altamente disponíveis.	42
4.8	Tempo médio de recuperação após a falha de uma única VNF.	46
4.9	Tempo médio de recuperação de múltiplas falhas por SFC.	47
4.10	Utilização de CPU.	48
4.11	Utilização de memória.	48
4.12	Vazão da SFC em um cenário sem falhas.	49
4.13	Vazão da SFC em um cenário com falhas.	49
5.1	Fluxo de execução da FIT-SFC..	57
5.2	Cenário de impossibilidade de quórum (falhas bizantinas).	58
5.3	Cenário de conluio com cliente malicioso.	58
5.4	A arquitetura FIT-SFC.	59
5.5	Comparação da latência do serviço de DNS.	62
5.6	Comparação da latência do serviço de DNS autenticado.	63
5.7	Comparação da latência do serviço de balanceamento de cargas.	63
6.1	Uma rede virtualizada genérica.	67
6.2	NFV-FD: arquitetura.	68
6.3	VNF-Consensus: arquitetura..	69
6.4	NFV-RBCast: arquitetura.	71
6.5	A arquitetura NFV-COIN.	71
6.6	Avaliação da latência para o serviço de cache.	79

6.7	Avaliação da vazão para o serviço de balanceamento de cargas.	80
-----	---	----

LISTA DE TABELAS

4.1	Comparação entre os diferentes tipos de mecanismos de resiliência.	41
4.2	Disponibilidade da SFC conforme o MTBF varia.	50
4.3	Comparação das soluções que fornecem alta disponibilidade para serviços e funções virtualizadas.	53
6.1	Serviços INC e NFV-COIN.	66
6.2	Exemplos de primitivas da NFV-COIN <i>Services</i> API.	72
6.3	Comparação da vazão considerando diferentes cenários.. . . .	77
6.4	Comparação da latência: <i>baseline</i> e NFV-COIN.	78

LISTA DE ACRÔNIMOS

API	<i>Application Programming Interface</i>
ASICs	<i>Application Specific Integrated Circuits</i>
CAPEX	<i>CAPital EXpenditures</i>
CIC	<i>Communication-Induced Checkpointing</i>
COIN	<i>COmputing In the Network</i>
CRIU	<i>Checkpoint/Restore in Userspace</i>
DNS	<i>Domain Name System</i>
EMS	<i>Element Management System</i>
ETSI	<i>European Telecommunications Standards Institute</i>
FCAPS	<i>Fault, Configuration, Accounting, Performance, and Security</i>
FD	<i>Failure Detector</i>
FIFO	<i>First-In First-Out</i>
FIT-SFC	<i>Fault- & Intrusion Tolerant SFC</i>
FMS	<i>Fault Management System</i>
FPGA	<i>Field Programmable Gate Array</i>
GST	<i>Global Stabilization Time</i>
IDS	<i>Intrusion Detection System</i>
IETF	<i>Internet Engineering Task Force</i>
INC	<i>In-Network Computing</i>
IoT	<i>Internet of Things</i>
ISG	<i>Industry Specification Group</i>
KVM	<i>Kernel-based Virtual Machine</i>
LB	<i>Load Balancer</i>
LLDP	<i>Link Layer Discovery Protocol</i>
LRU	<i>Least Recently Used</i>
MAC	<i>Message Authentication Code</i>
MPI	<i>Message Passing Interface</i>
MTBF	<i>Mean Time Between Failures</i>
NAT	<i>Network Address Translation</i>
NFV	<i>Network Function Virtualization</i>
NFVI	<i>NFV Infrastructure</i>
NFV-MANO	<i>NFV MANagement and Orchestration</i>
NFVO	<i>NFV Orchestrator</i>
NFVS	<i>NFV-COIN Supervisor</i>
NHAM	<i>NFV High Availability Module</i>

NSH	<i>Network Service Header</i>
OPEX	<i>Operational EXpenditures</i>
OSS/BSS	<i>Operations Support System/Business Support System</i>
PBFT	<i>Practical Byzantine Fault Tolerance</i>
PID	<i>Process Identifier</i>
QEMU	<i>Quick EMUlator</i>
REST	<i>REpresentational State Transfer</i>
SC	<i>Service Classifier</i>
SDN	<i>Software-Defined Network</i>
SFC	<i>Service Function Chain</i>
SFF	<i>Service Function Forwarder</i>
SFP	<i>Service Function Path</i>
SMR	<i>State Machine Replication</i>
TOSCA	<i>Topology and Orchestration Specification for Cloud Applications</i>
URL	<i>Uniform Resource Locator</i>
VIM	<i>Virtualized Infrastructure Manager</i>
VNF	<i>Virtualized Network Function</i>
VNFD	<i>VNF Descriptor</i>
VNFM	<i>VNF Manager</i>
VM	<i>Virtual Machine</i>
VSM	<i>VNF State Manager</i>

SUMÁRIO

1	INTRODUÇÃO	14
1.1	DEFINIÇÃO DO PROBLEMA.	15
1.2	CONTRIBUIÇÕES DA TESE	16
1.3	ORGANIZAÇÃO DA TESE	17
2	VIRTUALIZAÇÃO DE FUNÇÕES DE REDE	18
2.1	INTRODUÇÃO ÀS FUNÇÕES VIRTUALIZADAS DE REDE.	18
2.2	O PADRÃO ETSI PARA O GERENCIAMENTO E ORQUESTRAÇÃO DE FUNÇÕES VIRTUALIZADAS DE REDE.	19
2.3	SERVICE FUNCTION CHAINS.	21
3	CONFIANÇA NO FUNCIONAMENTO DE SISTEMAS DISTRIBUÍDOS	23
3.1	SISTEMAS DISTRIBUÍDOS TOLERANTES A FALHAS	23
3.2	REPLICAÇÃO DISTRIBUÍDA.	25
3.3	<i>ROLLBACK-RECOVERY</i> : DEFINIÇÕES BÁSICAS.	26
3.4	PBFT: <i>PRACTICAL BYZANTINE FAULT TOLERANCE</i> .	28
3.5	CONFIABILIDADE NA VIRTUALIZAÇÃO DE FUNÇÕES DE REDE.	28
3.5.1	Requisitos de Resiliência	29
3.5.2	Métodos de Resiliência	31
4	UMA ARQUITETURA DE ALTA DISPONIBILIDADE PARA SERVIÇOS VIRTUALIZADOS	33
4.1	EXPLORANDO OS DESAFIOS DE CONFIABILIDADE EM AMBIENTES NFV	33
4.2	NHAM: NFV <i>HIGH AVAILABILITY MODULE</i> .	35
4.2.1	Modelo de Sistema	35
4.2.2	A Arquitetura NHAM.	36
4.2.3	Gerenciando o Estado de VNFs <i>Stateful</i>	37
4.2.4	Mecanismos de Resiliência e Recuperação de Falhas	38
4.2.5	Comparação dos Mecanismos de Resiliência.	41
4.3	SERVIÇOS DE REDE ALTAMENTE DISPONÍVEIS	41
4.3.1	Gerenciamento de <i>Buffers</i> e Premissas	42
4.3.2	A Estratégia SFC <i>Hold/Release</i> .	43
4.4	IMPLEMENTAÇÃO E RESULTADOS EXPERIMENTAIS	44
4.4.1	Recuperação de Falhas	45
4.4.2	Utilização de Recursos Computacionais	47
4.4.3	Avaliando a Vazão das SFCs	48
4.4.4	Avaliando a Disponibilidade das SFCs	50

4.5	TRABALHOS RELACIONADOS	50
4.6	CONCLUSÕES PARCIAIS.	53
5	UMA ARQUITETURA PARA SFCS TOLERANTES A FALHAS E INTRUSÃO	55
5.1	FALHAS E INTRUSÕES EM COMPONENTES DA ARQUITETURA SFC . .	55
5.2	FIT-SFC: <i>FAULT- & INTRUSION- TOLERANT</i> SFC	56
5.2.1	Cenários com Falhas Bizantinas	57
5.2.2	Funcionamento da Arquitetura FIT-SFC	58
5.2.3	Garantindo a Consistência de VNFs Replicadas	60
5.3	IMPLEMENTAÇÃO E RESULTADOS EXPERIMENTAIS	60
5.3.1	Implementação da Arquitetura FIT-SFC e dos Serviços Virtualizados.	61
5.3.2	Resultados Obtidos	61
5.4	CONCLUSÕES PARCIAIS.	64
6	NFV-COIN: NFV <i>COMPUTING IN THE NETWORK</i>	65
6.1	<i>COMPUTING IN THE NETWORK</i> : UMA VISÃO GERAL.	65
6.2	SERVIÇOS NFV-COIN: TRÊS ESTUDOS DE CASO	67
6.2.1	Detector de Falha como um Serviço NFV-COIN: NFV-FD	68
6.2.2	Consenso como um Serviço NFV-COIN: VNF-Consensus	69
6.2.3	Difusão Confiável e Ordenada como um Serviço NFV-COIN: NFV-RBCast . . .	70
6.3	A ARQUITETURA NFV-COIN	70
6.3.1	NFV-COIN <i>Services</i> API	72
6.3.2	NFVS: NFV-COIN <i>Supervisor</i>	73
6.4	FUNCIONALIDADES DA ARQUITETURA NFV-COIN	74
6.4.1	Estabelecimento de Conexão com Serviços NFV-COIN	74
6.4.2	Comunicação com os Serviços NFV-COIN	75
6.4.3	Gerenciando Serviços NFV-COIN	75
6.5	IMPLEMENTAÇÃO E RESULTADOS EXPERIMENTAIS	75
6.5.1	Avaliação do Impacto na Latência e Vazão	77
6.5.2	Avaliação do Serviço de Cache	78
6.5.3	Avaliação do Serviço de Balanceamento de Cargas	79
6.6	CONCLUSÕES PARCIAIS.	80
7	CONCLUSÃO	81
	REFERÊNCIAS	84
	APÊNDICE A – PUBLICAÇÕES	92
A.1	TRABALHOS PUBLICADOS NO ÂMBITO DA TESE	92
A.2	TRABALHOS PUBLICADOS NO ÂMBITO DO GRUPO DE PESQUISA . . .	92

1 INTRODUÇÃO

Os protocolos da Internet foram desenvolvidos há décadas, e não foram projetados prevendo o extraordinário crescimento que ocorreu, o que contribuiu para a chamada “ossificação da Internet”. As tecnologias de virtualização representam uma estratégia concreta para a solução deste problema, permitindo a evolução da rede para múltiplas direções. A Virtualização de Funções de Rede (*Network Function Virtualization* - NFV) é uma das principais tecnologias de virtualização que permite tornar a rede altamente programável. Através de técnicas de virtualização, funções de rede (*e.g.*, *firewalls* e detectores de intrusão) tradicionalmente disponibilizadas através de *middleboxes*, podem ser substituídas por software executado em hardware genérico (Mijumbi et al., 2016; Chiosi et al., 2012).

No paradigma NFV, funções de rede são implementadas, disponibilizadas e executadas em dispositivos virtuais, denominados de Funções Virtualizadas de Rede (*Virtualized Network Functions* - VNFs) (Mijumbi et al., 2016). Dessa forma, as funções de rede que antes eram produzidas apenas por um número limitado de desenvolvedores, agora podem ser obtidas de maneira virtualizada em *marketplaces* na Internet (Bondan et al., 2019). Ainda, serviços de rede completos podem ser construídos através da combinação de múltiplas VNFs conectadas em uma ordem predefinida, denominada de SFC (*Service Function Chain*) (Halpern e Pignataro, 2015).

A adoção da tecnologia NFV representa um impacto significativo em termos de flexibilidade e gerenciamento da rede. Por executarem de maneira virtualizada, a criação, alteração e atualização de funções virtualizadas não necessitam da instalação de hardware especializado. Consequentemente, as despesas de capital (*CAPital EXpenditures* - CAPEX) e despesas operacionais (*OPerational EXpenditures* - OPEX) são reduzidas (Cotroneo et al., 2014a). Outra vantagem significativa das VNFs está na possibilidade de ajuste automático (*i.e.*, *autoscaling*) dos recursos disponibilizados para cada função virtualizada, já que é possível aumentar ou diminuir o uso de recursos computacionais de acordo com a demanda necessária, fazendo uso eficiente dos recursos disponíveis.

O instituto ETSI (*European Telecommunications Standards Institute*) vem propondo especificações para a padronização de uma arquitetura de referência NFV (ETSI, 2016). A arquitetura, denominada de NFV-MANO (*NFV MANAgement and Orchestration*) tem como objetivo principal facilitar a gerência e execução de serviços virtualizados, além de permitir o suporte à execução de VNFs de diferentes desenvolvedores (ETSI, 2015b; Quittek et al., 2014). Essa arquitetura também facilita a integração entre diferentes plataformas NFV e sistemas legados. A arquitetura NFV-MANO também realiza o controle do ciclo de vida tanto de VNFs quanto de SFCs, permitindo a criação, remoção e atualização de cada função.

Apesar das vantagens que a tecnologia NFV oferece, existem diversos desafios que devem ser investigados para permitir a sua ampla adoção e utilização efetiva. Em particular, esta Tese de Doutorado apresenta três contribuições em dois contextos principais. As duas primeiras contribuições estão relacionadas à tolerância a falhas de serviços virtualizados, enquanto que a última contribuição investiga a possibilidade de executar e gerenciar serviços virtualizados arbitrários no núcleo da rede.

A seguir é apresentada a definição do problema tratado na Tese e são descritas as principais contribuições, para finalmente ser apresentada a organização da Tese.

1.1 DEFINIÇÃO DO PROBLEMA

O primeiro desafio explorado nesta Tese de Doutorado está no fato de que serviços implementados e disponibilizados de maneira virtualizada são mais propensos a falhas em comparação com as alternativas executadas em hardware especializado (Cotroneo et al., 2019; Han et al., 2017, 2015; Gunawi et al., 2016, 2014). Em particular, a indisponibilidade de serviços virtualizados que hospedam plataformas de comércio online, transações bancárias, *streaming* de vídeo e armazenamento online podem gerar prejuízos monetários significativos (Gill et al., 2011; Gunawi et al., 2016; Sahoo et al., 2010b). Ainda, a falha em serviços de segurança podem levar ao uso não autorizado do sistema (Pattaranantakul et al., 2018), afetando não somente a infraestrutura subjacente (*i.e.*, rede NFV), como também os próprios serviços dos usuários. Por outro lado, funções de rede implementadas em hardware especializado são geralmente projetadas para atingir os níveis de alta disponibilidade exigidos pelos sistemas de telecomunicações: 99.999% (*five nines*) – porcentagem de tempo em que o serviço de rede estava acessível e funcional para o usuário (Gray e Siewiorek, 1991).

Vários trabalhos foram propostos para garantir a alta disponibilidade de serviços virtualizados (Cully et al., 2008; Kulkarni et al., 2018; Kablan et al., 2017; Li et al., 2015). No entanto, tais soluções em geral possuem limitações significativas, tais como a dificuldade de implantação em sistemas já existentes, necessidade de extensas modificações na estrutura do serviço, ou simplesmente por não fornecerem um conjunto completo de funcionalidades para garantir a alta disponibilidade de ponta-a-ponta do sistema. O problema torna-se ainda mais complexo ao considerar que grande parte das funções de rede são *stateful* – o estado interno da função muda de acordo com os pacotes processados. Assim, a recuperação de um serviço *stateful* só ocorre após a restauração do seu estado antes da falha.

O segundo desafio reside no fato de que, em geral, serviços virtualizados são executados sobre uma arquitetura subjacente. Tal arquitetura é normalmente composta por um conjunto de componentes internos que classificam e encaminham corretamente o tráfego pelo serviço em questão. No entanto, por executarem funcionalidades críticas para o funcionamento do serviço, a falha em qualquer destes componentes pode comprometer múltiplos serviços. Apesar de serviços virtualizados de rede serem amplamente explorados na literatura, diversos aspectos relacionados a tolerância a falhas ainda são marginalmente abordados.

Por fim, apesar de que a tecnologia NFV até então foi utilizada majoritariamente para a oferta de funções tradicionais de rede, o terceiro desafio investigado nesta Tese de Doutorado explora a possibilidade de utilizar o paradigma NFV para permitir que o próprio núcleo da rede ofereça serviços originais e inovativos. Com os múltiplos avanços nas tecnologias de virtualização, serviços de rede podem ser executados diretamente no núcleo da rede, no conceito que tem sido chamado de COIN (*COmputing In the Network*) (Zeng et al., 2021).

O principal objetivo do conceito COIN é unir a comunicação da rede com o processamento do tráfego dentro da rede (*i.e.*, *In-Network Computing* - INC), melhorando a integração entre múltiplas aplicações que exigem baixa latência, alta vazão e alta disponibilidade. Consequentemente, surge a oportunidade de unir o paradigma NFV no contexto COIN para oferecer serviços inovadores fornecidos nativamente pela própria rede. No entanto, a dificuldade está no fato de que a arquitetura de referência NFV-MANO não atende completamente a todos os requisitos para que serviços NFV-COIN sejam executados em larga escala em ambientes de produção.

1.2 CONTRIBUIÇÕES DA TESE

A primeira contribuição desta Tese propõe o NHAM (NFV *High Availability Module*), uma arquitetura NFV de alta disponibilidade para serviços virtualizados *stateful*, integrada como um módulo da arquitetura de referência NFV-MANO. O NHAM realiza o gerenciamento de falhas, monitorando cada serviço virtualizado, detectando falhas, e aplicando os procedimentos correspondentes de recuperação que garantam a recuperação correta de ponta-a-ponta de serviços *stateful*. A arquitetura proposta também inclui diferentes mecanismos de resiliência que podem ser aplicados de acordo com os requisitos de disponibilidade de cada serviço.

O NHAM é agnóstico em relação à tecnologia NFV: qualquer serviço virtualizado torna-se altamente disponível sem a necessidade de alterações durante a sua implementação, *i.e.*, tanto VNFs quanto SFCs simplesmente herdam as propriedades de alta disponibilidade de maneira transparente para o usuário. Para tanto, um componente é proposto para recuperar o estado interno de cada instância de VNF com técnicas baseadas em *Checkpoint/Restore* (Elnozahy et al., 2002). Além disso, para garantir que as falhas não impactem no funcionamento global do serviço, o NHAM incorpora estratégias de gerenciamento de *buffers* para garantir que o serviço seja recuperado corretamente, evitando a perda e/ou duplicação de pacotes.

Como prova de conceito, um protótipo da arquitetura NHAM foi implementado e experimentos mostram o desempenho e os níveis de disponibilidade atingidos de acordo com as diferentes estratégias de resiliência. Em especial, dependendo da estratégia e dos parâmetros, é possível alcançar a disponibilidade exigida dos sistemas comerciais de telecomunicações (*i.e.*, *five nines*).

A segunda contribuição desta Tese propõe a FIT-SFC (*Fault- & Intrusion- Tolerant SFC*): uma arquitetura para suportar serviços virtuais seguros e altamente disponíveis. Enquanto grande parte dos trabalhos da literatura consideram somente falhas por parada, a FIT-SFC utiliza estratégias de replicação para tolerar também falhas por omissão ou intrusão (*i.e.*, falhas bizantinas) de qualquer componente da arquitetura SFC.

Um protótipo da arquitetura foi implementado como prova de conceito e resultados experimentais são apresentados para os custos, em termos de latência para tolerar as falhas. Para tanto, três serviços virtualizados foram implementados e avaliados sobre a FIT-SFC: um serviço de DNS (*Domain Name System*), um serviço de DNS autenticado e um serviço de balanceamento de carga (*Load Balancer* - LB).

Por fim, a última contribuição desta Tese explora a sinergia entre NFV e COIN, no conceito denominado de NFV-COIN. Quando um serviço NFV-COIN é disponibilizado em uma rede, é essencial que este serviço seja instanciado e gerenciado de forma padronizada entre diferentes usuários, *i.e.*, sem exigir que o usuário tenha conhecimento sobre a estrutura interna e funcionamento da função. O objetivo é que diferentes usuários simplesmente importem em suas aplicações o serviço necessário através de primitivas de alto nível bem definidas.

Para tanto, uma arquitetura NFV-COIN é proposta, definindo como os serviços devem ser oferecidos, configurados, mantidos e acessados. Para permitir a instanciação e gerenciamento padronizado de serviços NFV-COIN, uma interface de comunicação entre o usuário e a rede NFV é proposta, possibilitando a comunicação entre aplicação e os serviços NFV-COIN. Além disso, é proposto um componente intermediário entre os usuários finais e os serviços NFV-COIN, responsável principalmente pela tradução de requisições de alto nível (originadas a partir do cliente) em requisições específicas do NFV-MANO.

Como prova de conceito, um protótipo da arquitetura foi implementado, além de dois serviços implementados no conceito NFV-COIN: um serviço de cache e um balanceador de cargas. Ambos os serviços são avaliados e mostram que a arquitetura NFV-COIN não introduz

sobrecargas significativas no desempenho ao passo em que permite a simplicidade e flexibilidade na execução de serviços de rede arbitrários.

1.3 ORGANIZAÇÃO DA TESE

O restante desta Tese de Doutorado está organizada da seguinte forma. O Capítulo 2 introduz os conceitos elementares de NFV, além de apresentar uma visão geral da arquitetura de referência NFV-MANO e a arquitetura interna IETF de SFCs. O Capítulo 3 apresenta conceitos fundamentais sobre a confiança no funcionamento de sistemas distribuídos, além de descrever algumas das técnicas para garantir a alta disponibilidade de serviços. O Capítulo 4 detalha o NHAM, descrevendo a sua arquitetura e componentes internos, bem como o protótipo implementado e resultados obtidos. Em seguida, o Capítulo 5 descreve a FIT-SFC, apresentando em detalhes as estratégias para tornar a arquitetura SFC tolerante a falhas bizantinas, além de sua implementação e avaliação. Por sua vez, o Capítulo 6 introduz o conceito NFV-COIN, apresentando também uma arquitetura que permite a criação, execução e gerenciamento de serviços virtualizados arbitrários na rede. São apresentados resultados experimentais obtidos a partir de um protótipo e serviços NFV-COIN implementados. Por fim, as conclusões e trabalhos futuros são apresentadas no Capítulo 7.

2 VIRTUALIZAÇÃO DE FUNÇÕES DE REDE

Neste capítulo os principais conceitos de Virtualização de Funções de Rede são apresentados. A Seção 2.1 descreve as motivações e as vantagens de utilizar tecnologia NFV. A Seção 2.2 apresenta e descreve a arquitetura de referência NFV-MANO proposta pela ETSI. Por sua vez, a Seção 2.3 introduz o conceito de SFC, detalhando os componentes da arquitetura proposta pela IETF.

2.1 INTRODUÇÃO ÀS FUNÇÕES VIRTUALIZADAS DE REDE

As redes de computadores utilizam diversos serviços, muitos dos quais são tradicionalmente implementados em *middleboxes* (e.g., *firewalls*, *Network Address Translation*, sistemas de detecção de intrusão) disponibilizados em dispositivos de hardware especializado (Sekar et al., 2012). Estes serviços representam uma importante fração das despesas de capital (*CAPital EXpenditures* - CAPEX) e despesas operacionais (*OPERational EXpenditures* - OPEX) de uma rede, uma vez que cada um desses componentes exige hardware especializado (Cotroneo et al., 2014a). Além disso, estes dispositivos são complexos para implantar, gerenciar e solucionar problemas, devido às suas instalações específicas e configurações manuais (Sherry et al., 2012). Por fim, os *middleboxes* não permitem o redimensionamento dinâmico de recursos de acordo com a demanda da rede, gerando potencialmente um desperdício dos recursos computacionais disponíveis no sistema, *i.e.*, se um *middlebox* está utilizando apenas metade dos seus recursos disponíveis, estes poderiam ser alocados para outra tarefa.

O paradigma NFV surge como uma alternativa concreta para flexibilizar e simplificar o projeto, desenvolvimento e gerência dos diversos serviços de rede (Chiosi et al., 2012). Através de técnicas de virtualização, funções de rede passam a ser implementadas e disponibilizadas por meio de dispositivos virtuais, denominados de Funções Virtualizadas de Rede (*Virtualized Network Function* - VNFs), executadas em hardware de prateleira. Dessa forma, um único hardware genérico pode hospedar múltiplos serviços, ao invés de adquirir um hardware específico para cada tipo de serviço necessário. A Figura 2.1 mostra uma comparação entre uma rede tradicional e um ambiente baseado em NFV. Na abordagem baseada em NFV, serviços de rede são implementados e executados através de uma camada de virtualização. Além disso, são mais fáceis de serem gerenciados e operados (Turchetti e Duarte, 2015).

A abordagem baseada em NFV possui diversas vantagens em comparação com as redes tradicionais. Por executarem de maneira virtualizada, a inclusão de novos serviços, bem como a alteração e atualização de serviços já existentes, não necessitam da instalação de novos equipamentos de hardware especializados, diminuindo custos e flexibilizando a implantação dos serviços (OPEX e CAPEX) (Mijumbi et al., 2016; Han et al., 2015). Consequentemente, a disponibilização de serviços também é facilitada: serviços virtualizados podem ser obtidos, por exemplo, através de *marketplaces* (Bondan et al., 2019).

Além da redução nos custos de aquisição e manutenção, as VNFs também utilizam menos energia e espaço físico do que os dispositivos de hardware dedicados (Han et al., 2015). Além disso, o gerenciamento da rede também é agilizado, na medida em que serviços de rede são facilmente criados, utilizados e removidos quando não forem mais necessários. Por fim, as VNFs podem ter seus recursos ajustados de maneira automática e dinâmica, aumentando ou diminuindo de acordo com a demanda necessária, fazendo uso eficiente dos recursos do sistema.

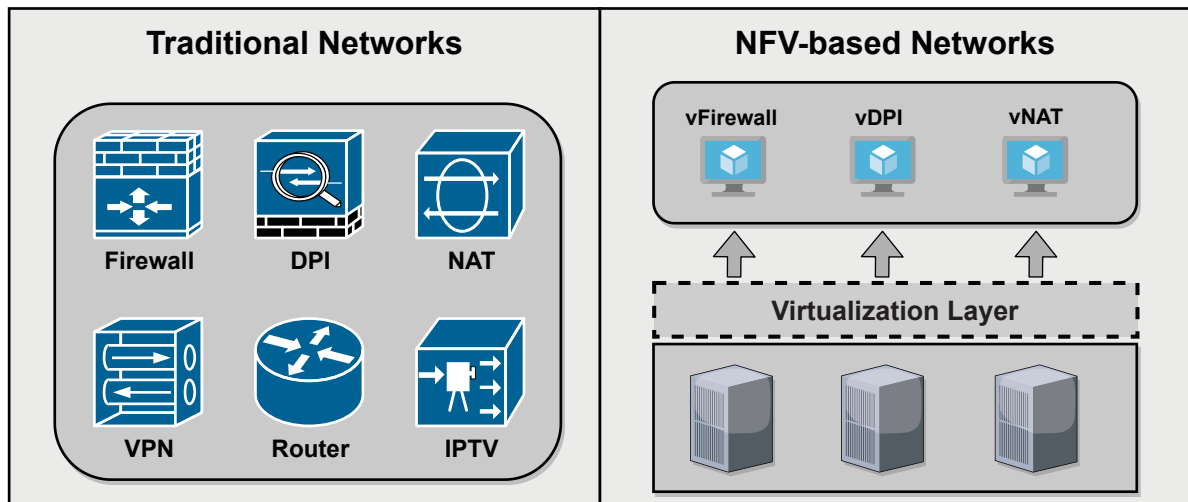


Figura 2.1: Comparação entre uma rede tradicional e uma rede baseada em NFV.

2.2 O PADRÃO ETSI PARA O GERENCIAMENTO E ORQUESTRAÇÃO DE FUNÇÕES VIRTUALIZADAS DE REDE

Com o objetivo de padronizar a execução e a gerência dos serviços baseados em NFV, além de permitir a interoperabilidade de uma grande variedade de VNFs provenientes de diferentes desenvolvedores, o grupo ISG (*Industry Specification Group*) criado pela ETSI (*European Telecommunications Standards Institute*) vem coordenando esforços para propor uma arquitetura de referência para NFV (Quittek et al., 2014). A Figura 2.2 mostra a arquitetura proposta pela ETSI, composta por três blocos principais: NFV-MANO (NFV *MAN*agement and *Or*chestration), NFVI (NFV *I*nfrastructure) e as VNFs propriamente ditas. Cada bloco é descrito a seguir.

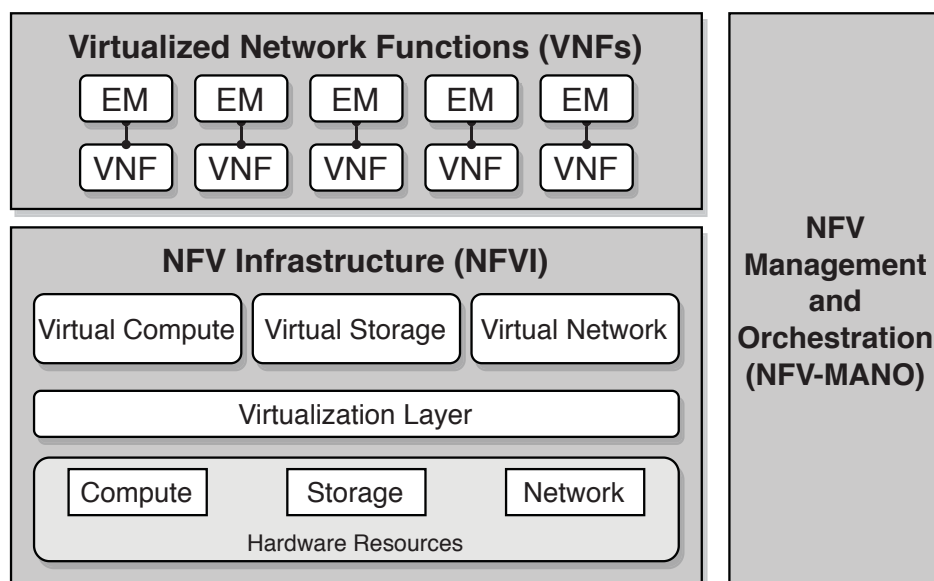


Figura 2.2: Arquitetura NFV alto nível proposta pela ETSI.

O bloco NFV-MANO é responsável principalmente pelo ciclo de vida, orquestração e gerenciamento dos serviços virtualizados. Além disso, o NFV-MANO fornece interfaces de comunicação padronizadas além de abstrair os recursos computacionais necessários para executar

as VNFs (ETSI, 2015b). As responsabilidades do NFV-MANO vão desde a criação de VNFs até a composição entre elas.

O segundo bloco ilustrado na Figura 2.2 é a NFVI, representando a infraestrutura virtualizada na qual as VNFs são instanciadas, gerenciadas e executadas. A parte física da NFVI consiste dos recursos computacionais responsáveis pelo processamento (*Compute*), armazenamento físico de dados (*Storage*) e de rede (*Network*). Os recursos físicos são abstraídos em recursos virtuais através de uma camada de virtualização (*Virtualization Layer*), composta por um *hypervisor* – componente responsável por criar e gerenciar dispositivos virtuais, tais como máquinas virtuais (*Virtual Machines* - VM) (Sahoo et al., 2010a) ou *containers* (Docker, 2017). Essa abordagem permite que cada VNF seja executada de maneira independente e isolada.

Por fim, o último bloco da arquitetura NFV-MANO corresponde às próprias VNFs. Este bloco representa as instâncias de cada dispositivo virtualizado responsável pela execução das funções de rede, executadas sobre a NFVI. De forma a especificar uma VNF em termos de requisitos operacionais e de implantação, definindo, por exemplo, configurações de rede (*e.g.*, endereço IP, interfaces de rede) e recursos computacionais (*e.g.*, CPU, RAM, disco), cada VNF possui um *VNF Descriptor* (VNFD) associado. Os VNFDs possuem informações que são utilizadas pelo NFV-MANO, como políticas para a realocação de recursos computacionais sob demanda e de monitoramento. Em geral, os VNFDs são baseados no padrão TOSCA (*Topology and Orchestration Specification for Cloud Applications*) (Li e Crandall, 2017). Além do VNFD, cada VNF deve fornecer um EMS (*Element Management System*), componente responsável pelo FCAPS (*Fault, Configuration, Accounting, Performance, and Security*) de cada função virtualizada. Este bloco inclui também as SFCs (*Service Function Chains*) – serviços de rede que consistem de múltiplas VNFs encadeadas.

Conforme descrito acima, o bloco NFV-MANO realiza a gerência e orquestração das funções virtualizadas de rede. A Figura 2.3 mostra os detalhes do bloco NFV-MANO, que pode ser dividido em três módulos principais: *NFV Orchestrator* (NFVO), *VNF Manager* (VNFM) e *Virtualized Infrastructure Manager* (VIM). Cada módulo é descrito a seguir.

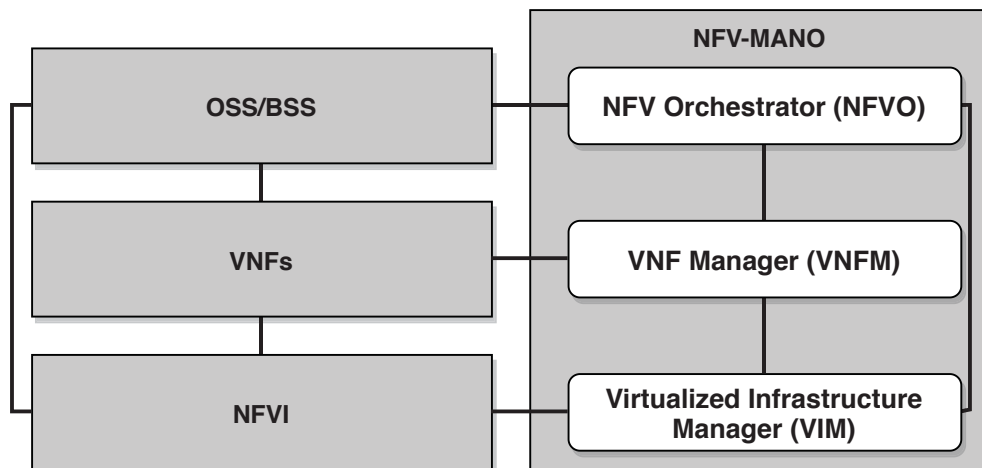


Figura 2.3: Componentes do NFV-MANO.

O primeiro módulo do NFV-MANO é o NFVO, responsável principalmente pelo gerenciamento e orquestração dos recursos alocados na infraestrutura virtualizada. O NFVO também gerencia o ciclo de vida dos serviços de rede coordenando a composição de VNFs para a criação de SFCs. Além disso, o NFVO realiza outras tarefas, como o aumento ou diminuição de recursos de uma VNF de acordo com a demanda (*i.e.*, *autoscaling*), gerenciamento de políticas, validação e autorização de requisições na NFVI, entre outras.

O módulo VNFM por sua vez tem como responsabilidade principal a realização da gerência do ciclo de vida das VNFs. A arquitetura prevê a utilização de um ou mais VNFMs, porém as operações de gerência devem ser suportadas por qualquer VNF, independente da sua implementação (ETSI, 2014). As principais atribuições deste módulo são: instanciação, remoção, atualização, configuração, atualização e escalonamento das VNFs, bem como o monitoramento dos recursos utilizados e notificações de falhas. O VNFM também deve ter acesso aos repositórios e às diferentes versões de VNFs, além de total conhecimento dos descritores (VNFDs), de forma a executar suas funcionalidades.

Por fim, o terceiro módulo do NFV-MANO é o VIM, responsável por controlar e gerenciar os recursos computacionais da infraestrutura virtualizada (*e.g.*, computação, armazenamento e rede). As principais tarefas executadas pelo VIM consistem da criação de dispositivos virtuais (*e.g.*, máquinas virtuais, *containers*), modificação dos recursos computacionais alocados, monitoramento da infraestrutura e gerenciamento das imagens virtuais (*i.e.*, *template* que contém o sistema operacional, arquivos e aplicações). O VIM geralmente corresponde a uma plataforma em nuvem (*e.g.*, OpenStack (OpenStack, 2019)). Assim como o VNFM, o NFV-MANO também prevê o uso de múltiplos VIMs, de forma que cada VIM gerencia uma infraestrutura virtualizada diferente.

Finalmente, apesar de não pertencer propriamente à arquitetura NFV-MANO, o módulo OSS/BSS (*Operations Support System/Business Support System*) representa os sistemas e aplicações externas que se comunicam com o ambiente NFV. Uma vez que o conceito de NFV recomenda o desenvolvimento de plataformas abertas, de forma a permitir a integração de soluções de diferentes desenvolvedores, os sistemas existentes não podem gerenciar as VNFs diretamente, visto que estas VNFs podem conter soluções proprietárias. Assim, o módulo OSS/BSS comunica-se com o NFV-MANO para realizar as requisições de gerência (*e.g.*, instanciar, remover, atualizar VNFs) através de interfaces padronizadas.

2.3 SERVICE FUNCTION CHAINS

Enquanto as VNFs executam funções específicas, serviços complexos de rede podem ser disponibilizados através da composição de múltiplas VNFs em uma SFC (Quittek et al., 2014). Uma SFC consiste de um conjunto de VNFs encadeadas em uma ordem predefinida, através do qual o tráfego é encaminhado (Halpern e Pignataro, 2015; Huff et al., 2018; Garcia et al., 2020).

Com o objetivo de flexibilizar a criação e gerenciamento de serviços, além de facilitar a configuração e criação de serviços em diferentes topologias, a IETF (*Internet Engineering Task Force*) propôs uma arquitetura padronizada para SFCs. De acordo com o IETF, a arquitetura de uma SFC consiste principalmente de elementos lógicos de classificação e de encaminhamento, tais como os *Service Classifiers*, *Service Function Forwarders* (SFFs) e as próprias VNFs (Halpern e Pignataro, 2015). A Figura 2.4 ilustra a arquitetura interna de uma SFC.

O elemento de classificação, denominado de *Service Classifier* (SC), é responsável por receber o tráfego de rede e determinar, baseando-se em um conjunto de políticas, para qual serviço (*i.e.*, SFC) o tráfego deve ser encaminhado. As políticas podem empregar diversos campos, tais como endereço IP de origem e destino, tipo de protocolo de comunicação (*e.g.*, TCP, UDP), porta de destino, entre outros.

Após um pacote ser classificado, o SC é responsável por encapsular este pacote, incluindo informações para formar o NSH (*Network Service Header*) (Quinn et al., 2018). Em particular, o NSH inclui informações que permitem o encaminhamento do pacote através do *Service Function Path* (SFP) – a ordem de VNFs da SFC através da qual o tráfego deverá percorrer (Halpern e Pignataro, 2015).

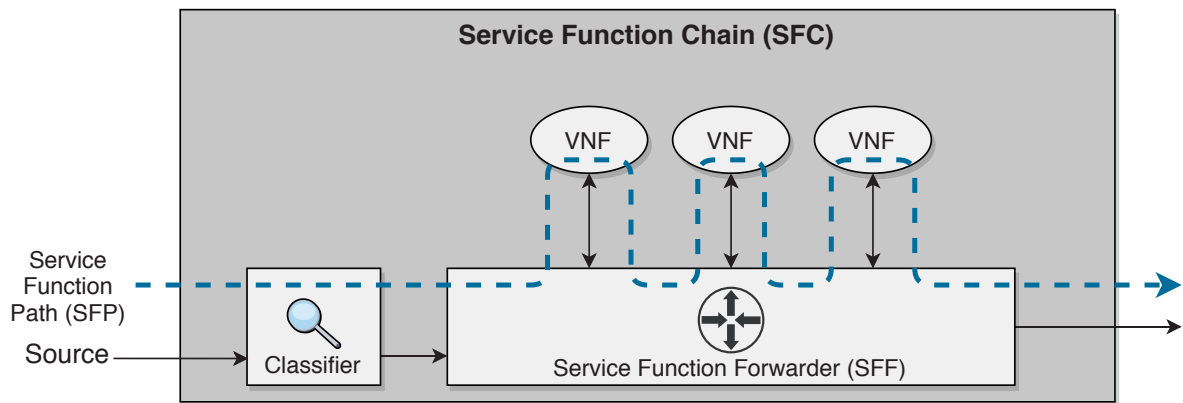


Figura 2.4: Arquitetura interna de uma SFC.

Uma vez que o tráfego foi classificado e encapsulado pelo SC, os pacotes são enviados ao SFF, componente responsável pelo encaminhamento propriamente dito dos pacotes de acordo com a ordem definida pelo SFP. O SFF utiliza as informações de encapsulamento inseridas pelo SC para enviar o tráfego para a primeira VNF da SFC. Após o recebimento e processamento dos pacotes, a VNF envia novamente o tráfego processado para a SFF. O SFF por sua vez recebe este tráfego, e realiza o encaminhamento para a próxima VNF de acordo com o SFP. Este processo se repete até que o tráfego tenha sido processado por todas as VNFs do SFP. Por fim, ao receber o tráfego processado pela última VNF da SFC, o SFF remove o encapsulamento de cada pacote feito pelo SC e envia o tráfego para o destino.

É importante observar que a arquitetura SFC permite que uma SFC seja composta por múltiplos SFFs. Dessa forma, se a quantidade de SFFs for maior do que a quantidade de VNFs, é possível que uma determinada VNF da SFC esteja acessível a partir de múltiplos SFFs. Neste caso, é responsabilidade do SFP determinar qual SFF deve realizar o encaminhamento das mensagens. Por outro lado, se a quantidade de SFFs for menor do que a quantidade de VNFs, alguns SFFs devem atender mais do que uma VNF. Ainda, existe a possibilidade de que um SFF seja compartilhado entre múltiplos serviços.

Apesar da arquitetura SFC descrita nesta seção não estar explicitamente no contexto de NFV, existem algumas formas de realizar esta integração. De acordo com a ETSI (ETSI, 2015a), uma das possibilidades é, com o auxílio de SDN (*Software-Defined Networks*) (Li e Chen, 2015), integrar a arquitetura SFC na própria NFVI. Neste cenário, um controlador SDN é incorporado a NFVI e conectado ao VIM, permitindo o encadeamento do tráfego através da SFC. Portanto, os *switches* e roteadores SDN na NFVI assumem as funcionalidades do SFF.

3 CONFIANÇA NO FUNCIONAMENTO DE SISTEMAS DISTRIBUÍDOS

O objetivo deste capítulo é descrever os conceitos fundamentais de sistemas distribuídos e tolerância a falhas utilizados neste trabalho. A Seção 3.1 apresenta as definições elementares de sistemas distribuídos e de tolerância a falhas, enquanto que as Seções 3.2 e 3.3 e 3.4 descrevem algumas das técnicas mais importantes para garantir alta disponibilidade em sistemas distribuídos. Por fim, a Seção 3.5 discute alguns conceitos importantes de confiabilidade no contexto de redes NFV.

3.1 SISTEMAS DISTRIBUÍDOS TOLERANTES A FALHAS

Um sistema distribuído é definido neste trabalho como um conjunto de processos (*i.e.*, programas em execução) que executam em máquinas distintas, cooperando entre si para resolver uma determinada tarefa através da troca de mensagens (Kshemkalyani e Singhal, 2011). Os problemas resolvidos em um sistema distribuído variam desde uma aplicação específica (*e.g.*, aplicações *web*, banco de dados distribuídos e sistemas operacionais distribuídos), até aplicações de propósito geral (*e.g.*, MPI – *Message Passing Interface* (Graham et al., 2005), JGroups (Ban et al., 2002), Zookeeper (Hunt et al., 2010)).

Em sistemas distribuídos é essencial considerar o comportamento dos processos com relação ao tempo, em especial, analisando os limites de tempo de resposta dos processos no sistema. Dessa forma, é possível assumir vários modelos temporais, tais como o modelo síncrono, assíncrono e os diversos modelos parcialmente síncronos (Mullender et al., 1993; Cachin et al., 2011), descritos a seguir. No modelo síncrono, existe um limite superior conhecido em que um processo irá executar uma determinada tarefa. Esta tarefa pode envolver tanto o recebimento e/ou a transmissão de mensagens quanto o processamento local dos dados. Por outro lado, o modelo assíncrono não faz nenhuma restrição com relação aos limites temporais, sendo caracterizado pela ausência de conhecimento temporal de qualquer ordem. Por fim, os modelos parcialmente síncronos definem níveis intermediários de sincronismo, *i.e.*, entre o síncrono e o assíncrono. Na prática, uma possível maneira de se obter um modelo parcialmente síncrono é utilizando *timeouts* com precisão – parâmetros utilizados para o controle do tempo, estabelecendo um tempo máximo de resposta para o recebimento da mensagem.

Uma aplicação deve executar as suas tarefas de acordo com a sua especificação. No entanto, falhas podem causar comportamentos inconsistentes, gerando possivelmente resultados incorretos. Dessa forma, é essencial garantir que o sistema continue operacional, mesmo com a falha de alguns de seus componentes ou processos. Por esse motivo, os sistemas devem satisfazer a propriedade de tolerância a falhas, garantindo, em qualquer circunstância, a execução correta do sistema (Pradhan, 1996; Avizienis et al., 2004). Para que um sistema seja tolerante a falhas, é necessário ter confiança no seu funcionamento (*dependability*). Para tanto, é necessário satisfazer os seguintes requisitos (Avizienis et al., 2004):

- Disponibilidade (*availability*): capacidade de atender os usuários autorizados que solicitam os seus serviços.
- Confiabilidade (*reliability*): capacidade de oferecer serviço correto e contínuo (*i.e.*, sem interrupções).
- Segurança (*safety*): capacidade de evitar consequências desastrosas.

- Integridade (*integrity*): capacidade de evitar a alteração não autorizada no sistema.
- Manutenção (*maintainability*): facilidade de sofrer alterações e modificações.

De maneira a garantir estas propriedades, os sistemas distribuídos devem ser capazes de permanecer operacionais mesmo após a falha de seus processos. Segundo a definição de *failure* (falha no serviço), esta pode ser identificada através de uma saída incorreta do sistema, sendo provocada por um ou mais erros. Erros são estados do sistema que decorrem de manifestações de *faults* (falhas em um ou mais processos do sistema). Uma vez que existem diversos tipos de falhas, vários modelos de falha foram propostos de maneira a especificar o modo pelo qual os processos do sistema podem falhar. Os modelos de falha são particularmente importantes, visto que os algoritmos utilizados para resolver uma determinada tarefa variam de acordo com o modelo de falha adotado e também o modelo temporal (Kshemkalyani e Singhal, 2011). Os principais modelos de falha são descritos a seguir.

O modelo de falhas por parada (*crash*) especifica a parada permanente de um processo, não executando nenhum processamento local, nem o envio de mensagens para outros processos. É importante ressaltar que, após a falha, o processo nunca se recupera. Por sua vez, a falha por *omissão* especifica que os processos enviam e/ou recebem apenas parte das mensagens que deveriam produzir. Por exemplo, essa falha é causada por erros de *buffer overflow* ou congestionamento da rede (Cachin et al., 2011). Já na falha bizantina o processo apresenta um comportamento arbitrário, produzindo respostas incongruentes com a sua especificação. A causa desse comportamento pode variar desde erros na implementação do sistema até ações maliciosas. A falha bizantina, por ser a mais abrangente em sua definição, engloba as falhas por parada e por omissão.

Existem ainda variações dos modelos descritos acima. Por exemplo, é possível considerar a recuperação de um processo após a sua falha. Nesse modelo, denominado de *crash-recovery*, o estado interno de um processo é armazenado em memória não volátil, permitindo a restauração das suas informações após a recuperação de uma falha por parada. O modelo *fail-stop* assume que os processos também falham por parada, porém garante que os demais processos corretos detectam a falha imediatamente. Por fim, a falha de temporização ocorre quando o processo não respeita os limites de tempo para o envio de mensagens. Esse tipo de falha ocorre apenas em sistemas que adotam o modelo síncrono, uma vez que no modelo assíncrono não assume nenhuma premissa temporal.

Apesar da susceptibilidade a falhas nos sistemas distribuídos, é essencial garantir a alta disponibilidade dos serviços, de forma a minimizar o tempo de inatividade e manter as propriedades de disponibilidade e confiabilidade. Uma vez que a disponibilidade de um serviço aumenta conforme o tempo total de recuperação diminui, diversos aspectos devem ser considerados, incluindo a detecção e recuperação consistente de falhas. A disponibilidade de um serviço pode ser representada de acordo com a seguinte equação (Piedad e Hawkins, 2001):

$$\text{Disponibilidade} = \frac{\text{Tempo Total Operacional}}{(\text{Tempo Total Operacional} + \text{Tempo Total de Recuperação})}$$

Portanto, a disponibilidade de um serviço pode apenas aumentar com a redução no tempo total de recuperação. Em especial, é necessário detectar e reagir a falhas de maneira eficiente, minimizando a indisponibilidade do serviço. O tempo médio entre a ocorrência de falhas (*Mean Time Between Failures* - MTBF) é uma medida utilizada para especificar o tempo previsto até que uma falha aconteça (Kshemkalyani e Singhal, 2011). Naturalmente, quanto maior o MTBF, maior será a disponibilidade do serviço.

Entre as diversas técnicas desenvolvidas para garantir a tolerância a falhas e a alta disponibilidade em sistemas distribuídos, as próximas seções descrevem a replicação distribuída e alguns dos algoritmos de *rollback-recovery*.

3.2 REPLICAÇÃO DISTRIBUÍDA

Uma das técnicas mais utilizadas para a construção de serviços tolerantes a falhas é a replicação (Charron-Bost et al., 2010). A Replicação de Máquina de Estados (*State Machine Replication* - SMR) (Lamport, 2019; Schneider, 1990) replica o estado de um processo em múltiplas réplicas (*i.e.*, cópias), garantindo que a falha de um ou mais processos (*i.e.*, servidores) não comprometa o funcionamento correto do serviço. A técnica de SMR consiste em garantir que toda sequência de operações é enviada a todas as réplicas exatamente na mesma ordem. Desta forma, a partir de um mesmo estado inicial, todas as réplicas sempre estarão no mesmo estado. É importante notar que a execução de cada réplica é feita de maneira determinística (Schneider, 1990).

Para atingir tais garantias, é necessário antes explorar o conceito de difusão confiável. A difusão confiável é uma das abstrações mais importantes para o desenvolvimento de sistemas distribuídos. Um processo de origem pode utilizar a difusão confiável para enviar uma mensagem de forma a garantir que todos os processos corretos na rede entreguem esta mensagem, mesmo que o processo de origem falhe. No entanto, somente a difusão confiável não é suficiente. Uma das formas de garantir a consistência para a SMR é através de uma primitiva de comunicação conhecida como difusão atômica (Hadzilacos e Toueg, 1993; Charron-Bost et al., 2010). A difusão atômica, em inglês *atomic broadcast*, ou *atomic reliable broadcast*, ou ainda *total-order broadcast*, garante que todos os processos entregam todas as mensagens exatamente em uma mesma ordem (Défago et al., 2004).

Uma forma de implementar a difusão atômica é através do consenso executado pelos processos replicados sobre a ordem de entrega das mensagens (Cachin et al., 2011). O consenso permite que todos os processos de um sistema distribuído cheguem a uma decisão comum (Fischer et al., 1985a). Processos de um sistema distribuído frequentemente precisam entrar em acordo em relação às ações executadas dentro do sistema. Informalmente, um algoritmo de consenso garante que todos os processos do sistema entrem em acordo sobre um único valor dentro de um conjunto de valores, possivelmente diferentes, propostos inicialmente pelos processos participantes (Borran et al., 2012). Além disso, o consenso deve ser garantindo mesmo com a falha de alguns de seus processos.

O Paxos é um algoritmo clássico de consenso tolerante a falhas (Lamport, 1998, 2001). No Paxos, os processos podem assumir os seguintes papéis: *proposers*, *acceptors* e *learners*. Os *proposers* propõem um valor, os *acceptors* escolhem um valor e os *learners* aprendem o valor decidido. Um único processo pode assumir qualquer um destes papéis ou múltiplos papéis simultaneamente. O algoritmo Paxos é ótimo em termos de resiliência (Lamport, 2006): para tolerar f falhas ele requer $2f + 1$ *acceptors*. Outro algoritmo de consenso é o Raft (Ongaro e Ousterhout, 2014), semelhante ao Paxos em termos de tolerância a falhas e desempenho, porém projetado para ser mais fácil de entender e implementar, uma vez que os principais componentes do consenso são separados em problemas isolados.

Em 1985, no entanto, provou-se que é impossível a execução do consenso em sistemas distribuídos assíncronos sujeitos a falhas do tipo *crash*. Essa impossibilidade é conhecida como impossibilidade FLP, descrita em (Fischer et al., 1985b). A raiz desta impossibilidade é o fato do modelo síncrono não refletir a realidade da maioria dos sistemas distribuídos reais, uma vez que não é uma tarefa trivial conhecer perfeitamente todos os seus limites de tempo. Apesar de

que os modelos assíncronos refletem melhor os sistemas reais, não é possível fazer nenhuma afirmação com relação ao tempo de resposta dos processos neste modelo. Isso implica, por exemplo, em que não se pode distinguir um processo falho de um processo lento. Sem essa distinção, o consenso fica sujeito à impossibilidade FLP.

Inspirado nesse impasse, a abstração de detectores de falhas é proposta em (Chandra e Toueg, 1996). Nessa abstração, cada processo tem acesso a um módulo local que possui informações sobre quais processos estão falhos e quais estão corretos. Essas informações são obtidas através do monitoramento dos processos do sistema. São utilizados *timeouts* para determinar se um processo é classificado como falho ou correto.

Por outro lado, as informações obtidas pelos detectores de falhas podem ser incorretas. Por exemplo, um processo falho pode ser considerado correto. Esse tipo de erro pode acontecer, por exemplo, se um processo falha logo após o detector coletar o estado atual do processo. Outra informação incorreta que um detector pode obter é considerar um processo correto como falho. Isso pode acontecer quando um processo está correto, porém está mais lento que os demais, atrasando a resposta por um tempo indeterminado. Além disso, como não é garantido nenhum limite de tempo nas respostas dos processos, detectores de falhas de processos diferentes podem ter visões diferentes, *i.e.*, um detector considera um processo como falho ao passo que outro detector considera o mesmo processo como correto. Dessa forma, os detectores de falhas são considerados como não confiáveis (Chandra e Toueg, 1996).

Em (Chandra e Toueg, 1996) os autores ainda classificam os detectores de falhas em várias classes, que variam de acordo com as propriedades de completude – todos os processos que estão em um estado falho são suspeitos – e de precisão – nenhum processo correto é suspeito. Uma destas classes permite resolver o consenso em sistemas assíncronos sujeitos à falha *crash* (Chandra et al., 1996), que, como dito acima, é usado para implementar a difusão atômica e a Replicação Máquina de Estados.

3.3 ROLLBACK-RECOVERY: DEFINIÇÕES BÁSICAS

Uma técnica muito utilizada na construção de sistemas distribuídos tolerantes a falhas é o *Rollback-Recovery* (Elnozahy et al., 2002; Koo e Toueg, 1987; Randell et al., 1978). Nesta técnica, informações são coletadas periodicamente sobre cada processo do sistema e armazenadas em memória não volátil, ou em um dispositivo considerado confiável, *i.e.*, que não é afetado por falhas. O *Rollback-Recovery* permite que, em casos de falha, os processos reiniciem as suas execuções a partir de um estado intermediário, ao invés de reiniciar completamente a sua execução, voltando ao estado inicial. Consequentemente, o processamento perdido em decorrência de uma falha é reduzido (Elnozahy et al., 2002). O principal objetivo do *Rollback-Recovery* é garantir que em casos de falha o sistema possa voltar a um estado global consistente. Um estado global pode ser definido como um conjunto dos estados locais dos seus processos (Kshemkalyani e Singhal, 2011). De acordo com (Elnozahy et al., 2002), os algoritmos de *Rollback-Recovery* podem ser classificados como baseados em *checkpoints* (*checkpoint-based*) ou baseados no registro de mensagens (*log-based*), ambos descritos a seguir.

Para prover a tolerância a falhas, os algoritmos baseados em *checkpoints* utilizam *checkpoints* – representações do estado interno de um processo (Koo e Toueg, 1987). O estado interno de um processo inclui o conteúdo dos registradores do processador, memória local, entre outras informações que podem depender do contexto de sua aplicação (Kshemkalyani e Singhal, 2011). Em caso de falha, o estado do sistema retorna para o conjunto mais recente de *checkpoints* consistentes, também conhecido como linha de recuperação (*recovery line*). Nessa abordagem,

os *checkpoints* podem ser coordenados (*coordinated*), não coordenados (*uncoordinated*) ou induzidos pela comunicação (*communication-induced checkpointing* – CIC), descritos a seguir.

Nos *checkpoints* não coordenados, cada processo executa os *checkpoints* de maneira independente. Apesar de ser uma abordagem simples de implementar, ela possui algumas desvantagens. Por exemplo, quando existem mensagens dependentes entre processos durante a execução sem falha, em caso de falha, as dependências forçam alguns processos que não falharam a também realizar o *rollback*. Em alguns cenários, esse processo de recuperação pode fazer com que os processos voltem até o início da sua execução, resultando na perda de todo o processamento até o momento antes da falha. Esse fenômeno é conhecido como efeito dominó (*domino effect*) (Elnozahy et al., 2002). Além disso, podem ocorrer a criação de *checkpoints* inúteis (*i.e.*, *checkpoints* que não fazem parte do estado global) e a necessidade de criar estratégias de *garbage collection* para remover *checkpoints* que não são mais necessários.

De forma a evitar o efeito dominó, os *checkpoints* coordenados exigem a sincronização entre os processos para formar um estado global consistente (Kshemkalyani e Singhal, 2011). Em casos de falha, o sistema pode sempre voltar ao estado consistente mais recente. Os *checkpoints* coordenados também evitam o armazenamento de múltiplos *checkpoints*. Em geral, os algoritmos baseados em *checkpoints* coordenados utilizam mensagens específicas para realizar as suas ações, tais como a criação de um novo *checkpoint* ou o *rollback* para um estado anterior. A maior desvantagem no *checkpoint* coordenado está no custo para garantir a sincronização entre os processos. Uma das maiores sobrecargas impostas está na latência, uma vez que toda a comunicação precisa aguardar enquanto o *checkpoint* coordenado é criado.

Por sua vez, os *checkpoints* induzidos pela comunicação evitam o efeito dominó sem a necessidade de coordenação dos *checkpoints*. Para tanto, dois tipos de *checkpoints* são gerados: *checkpoints* locais, gerados independentemente e *checkpoints* forçados, responsáveis por garantir a progressão da linha de recuperação. Nessa abordagem, ao invés de utilizar mensagens específicas para a realização dos *checkpoints*, informações são inseridas nos pacotes que são posteriormente utilizadas para decidir se um processo deve forçar um *checkpoint*. Essa abordagem, no entanto, pode gerar um grande número de *checkpoints* (Kshemkalyani e Singhal, 2011; Elnozahy et al., 2002).

Uma vez que a execução de um processo pode ser modelada como uma sequência de intervalos de estados determinísticos, os algoritmos baseados em registro de mensagens combinam o *checkpoint* com o registro de eventos não-determinísticos (Kshemkalyani e Singhal, 2011; Elnozahy et al., 2002). Durante a sua execução, cada processo identifica todos os seus eventos não-determinísticos (*e.g.*, recebimento de mensagens) e, para cada evento, armazena toda a informação necessária para a sua recuperação em uma abstração chamada de determinante. Além disso, cada processo realiza periodicamente *checkpoints*, a fim de reduzir a quantidade de *rollback*. Em caso de falha, o processo pode voltar ao *checkpoint* mais recente e reexecutar os determinantes salvos para retomar exatamente o seu estado antes da falha. Além disso, estes algoritmos garantem que após a recuperação do sistema não existam processos órfãos – processos cujo estado depende de um evento não-determinístico e que não pode ser reproduzido durante a recuperação.

As técnicas de *Rollback-Recovery* são extensas e podem ser aplicadas em diversas áreas. No contexto deste trabalho, consideramos apenas os algoritmos de *Rollback-Recovery* para sistemas distribuídos baseados em *checkpoints*.

3.4 PBFT: PRACTICAL BYZANTINE FAULT TOLERANCE

Como visto na Seção 3.1, uma falha bizantina decorrente de, por exemplo, um ataque malicioso, pode levar um nodo ou processo a apresentar um comportamento arbitrário. Neste contexto, algoritmos que toleram falhas bizantinas são essenciais para garantir o uso correto e contínuo de um sistema. Um destes algoritmos é o PBFT (*Practical Byzantine Fault Tolerance*) (Castro et al., 1999; Castro e Liskov, 2002). O PBFT é um algoritmo de replicação de máquina de estados capaz de tolerar falhas bizantinas. O objetivo deste algoritmo é garantir que todos os nodos corretos de um sistema decidam por um mesmo valor, mesmo que f nodos possuam falhas bizantinas.

O PBFT considera um sistema distribuído assíncrono e os nodos do sistema podem apresentar comportamento arbitrário, falhando ao entregar mensagens, atrasando ou duplicando as mensagens, ou entregando as mensagens fora de ordem. O algoritmo PBFT necessita de $3f + 1$ réplicas de um determinado componente para tolerar f falhas bizantinas. A razão de utilizar essa quantidade de réplicas é descrita a seguir.

Inicialmente, deve ser possível executar uma determinada requisição sem a necessidade de aguardar por f réplicas, uma vez que estas réplicas podem estar falhas. Ou seja, deve ser possível chegar a uma decisão após comunicar com $n - f$ réplicas. No entanto, existe a possibilidade de que estas f réplicas consideradas falhas, podem não estar falhas, mas simplesmente lentas. Consequentemente, das $n - f$ réplicas que responderam, f réplicas podem ser falhas. Portanto, para garantir que exista um quórum de réplicas corretas, é necessário que $n - 2f > f$, i.e., $n > 3f$.

No algoritmo PBFT, considera-se uma das réplicas como primária (também chamada de líder) e as demais réplicas como *backups*. O PBFT exige que as réplicas sejam determinísticas – a execução de um conjunto de mensagens sempre leva ao mesmo resultado. Além disso, as réplicas devem iniciar sempre no mesmo estado. No algoritmo PBFT as mensagens são autenticadas.

O protocolo de funcionamento do algoritmo PBFT é dividido em três fases: *pre-prepare*, *prepare* e *commit*. O funcionamento básico do algoritmo para o caso de não haver falhas é ilustrado na Figura 3.1 e é descrito a seguir. Inicialmente, o cliente envia uma mensagem m para a réplica primária. Em seguida, a fase *pre-prepare* inicia ao passo em que a réplica primária envia uma mensagem PRE-PREPARE para todas as réplicas *backups*, inserindo um número de sequência n na mensagem m . Por sua vez, cada réplica *backup* aceita a mensagem PRE-PREPARE se e somente se a mensagem da réplica primária for autenticada e nenhuma outra mensagem PRE-PREPARE foi aceita com o mesmo número de sequência n .

No momento em que uma réplica *backup* aceita uma mensagem PRE-PREPARE, inicia-se a fase *prepare*. A réplica então envia a mensagem PREPARE com a mensagem m e o número de sequência n para todas as demais réplicas. Ao receber $2f + 1$ mensagens PREPARE idênticas (i.e., mesmo m e mesmo n), a réplica *backup* aceita a mensagem e a fase *commit* inicia. Nesta fase, cada réplica envia novamente uma mensagem COMMIT de confirmação para todas as réplicas, contendo m e n . Quando uma réplica recebe $2f + 1$ mensagens de confirmação com o mesmo m e n , ela confirma que m é a n -ésima requisição a ser executada.

Em seguida, cada réplica executa a mesma requisição e, finalmente, envia uma resposta ao cliente. Por sua vez, o cliente aguarda por $f + 1$ respostas idênticas para a requisição.

3.5 CONFIABILIDADE NA VIRTUALIZAÇÃO DE FUNÇÕES DE REDE

Garantir as propriedades de disponibilidade e confiabilidade de um sistema é de extrema importância para a construção de serviços altamente disponíveis. Neste contexto, a ETSI vem

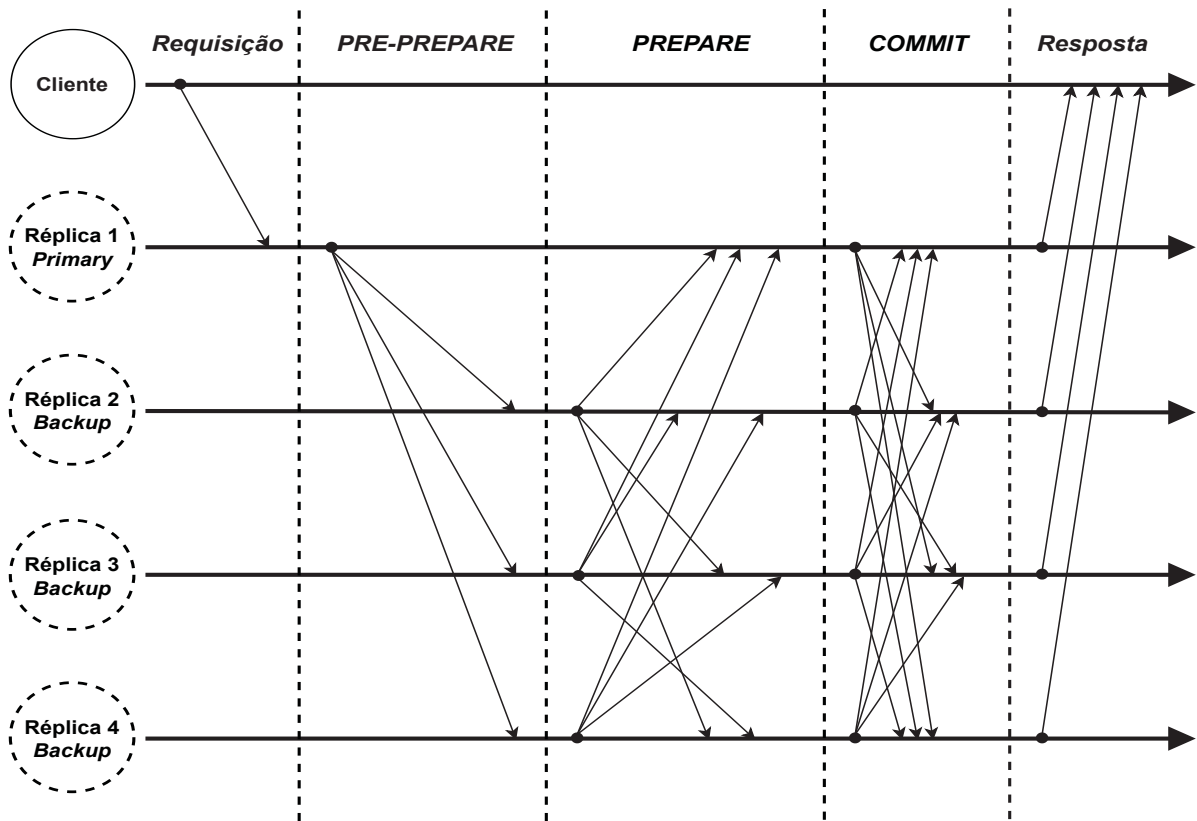


Figura 3.1: Execução do algoritmo PBFT.

definindo diversos requisitos de resiliência para ambientes e plataformas NFV (Schöller et al., 2015; Nakamura et al., 2016; Lac et al., 2017). No contexto de NFV, o termo resiliência (amplamente adotado nas definições da ETSI), se refere à habilidade da retomada do serviço para um nível normal, ou pelo menos aceitável, após a ocorrência de uma ou mais falhas (Lac et al., 2017). Em particular, uma plataforma NFV deve oferecer suporte à alta disponibilidade e resiliência para VNFs de diferentes tipos e desenvolvidas por diferentes fornecedores.

Considerando também a grande variedade das funções de rede, espera-se que diferentes níveis de resiliência possam ser definidos, uma vez que as VNFs possuem requisitos distintos. Por exemplo, funções de rede que processam tráfego em tempo real possuem requisitos mais restritos de resiliência do que funções que toleram um maior tempo de recuperação. Dessa forma, é essencial que a plataforma NFV ofereça suporte a diferentes níveis de resiliência, com diferentes estratégias e vantagens. Além disso, a plataforma NFV deve permitir que os próprios desenvolvedores das VNFs definam seus requisitos de resiliência.

A Seção 3.5.1 descreve em detalhes os requisitos de resiliência para NFV definidos pela ETSI, enquanto que a Seção 3.5.2 classifica os diversos métodos de redundância para garantir a continuidade do serviço em casos de falha.

3.5.1 Requisitos de Resiliência

Em geral, para que as VNFs em um ambiente virtualizado sejam altamente disponíveis, é necessário que as plataformas NFV atendam os seguintes requisitos (Schöller et al., 2015):

1. Devem permitir especificar e garantir requisitos de disponibilidade e confiabilidade;
2. Devem prover gerenciamento de falhas;

3. Devem permitir a recuperação automatizada em casos de falha;
4. Devem evitar pontos únicos de falha;
5. Devem suportar VNFs de diferentes fornecedores;

De acordo com a ETSI, para satisfazer os três primeiros requisitos (disponibilidade e confiabilidade, gerenciamento de falhas e recuperação automatizada), a plataforma NFV deve oferecer os seguintes mecanismos: (i) posicionamento de VNFs levando em conta aspectos de posicionamento baseado em afinidade (*affinity aware VNF placement*); (ii) proteção do estado interno da VNF e; (iii) detecção, prevenção e recuperação de falhas. Todos os mecanismos são descritos a seguir.

Posicionamento baseado em afinidade. Um dos serviços ofertados pelo VIM na arquitetura NFV-MANO é o posicionamento de VNFs, cujo objetivo está em escolher, de maneira otimizada, o melhor local (*e.g.*, rede, nodo, servidor) para instanciar uma VNF. A escolha é feita de acordo com um conjunto de restrições, tais como a disponibilidade de recursos ou a latência (Cappanera et al., 2019).

Do ponto de vista da resiliência, é possível incluir nas restrições regras de afinidade ou anti-afinidade. As regras de afinidade podem, por exemplo, definir que as VNFs de um mesmo grupo ou SFC devem ser instanciadas em um mesmo servidor físico. Neste cenário, falhas físicas no servidor (*e.g.*, falha no enlace de comunicação, falta de memória ou espaço de armazenamento) possivelmente comprometem todo o serviço, uma vez que todas as VNFs são executadas na mesma máquina. Por outro lado, as regras de anti-afinidade permitem definir que as VNFs de um mesmo grupo ou SFC sejam instanciadas em servidores diferentes, eliminando o problema de ponto único de falha. No entanto, as regras de anti-afinidade podem provocar uma maior latência no serviço, visto que as VNFs estão fisicamente distribuídas.

Proteção do estado interno da VNF. No contexto de funções de rede, uma VNF pode ser classificada como *stateless* – o estado interno está desacoplado do seu processamento – ou *stateful* – o estado interno da função de rede muda de acordo com os pacotes processados e a própria execução da função. Visto que grande parte das VNFs são *stateful* (Han et al., 2017), é essencial garantir que a plataforma NFV forneça mecanismos para garantir a disponibilidade e confiabilidade do seu serviço.

Existem várias possibilidades a serem exploradas no desenvolvimento de mecanismos para garantir a recuperação do estado interno das VNFs em caso de falha. Por exemplo, o estado interno pode ser armazenado em um dispositivo externo à VNF (*e.g.*, banco de dados) ou até mesmo diretamente em outra instância redundante da VNF. É importante notar que a proteção do estado independe da técnica utilizada, *i.e.*, todas as técnicas descritas no Capítulo 3 são aplicáveis. As diferentes estratégias de como o estado pode ser replicado serão descritas na Seção 3.5.2.

Detecção, prevenção e recuperação de falhas. Em geral, um sistema de gerenciamento de falhas consiste da detecção, prevenção e recuperação de falhas. A detecção de falhas em ambientes virtualizados pode ser feita em diferentes níveis: na própria aplicação, na própria VNF (*i.e.*, diretamente no *container* ou máquina virtual), ou no servidor que hospeda as VNFs. Dessa forma, as falhas que ocorrem em um ambiente baseado em NFV podem ser de diferentes tipos, tais como falhas no *software* (*i.e.*, na aplicação), no sistema operacional, ou falhas no *hardware* (*e.g.*, recursos computacionais, enlaces de comunicação).

Uma vez que falhas ocorrem em diferentes níveis, é necessário determinar quais os tipos de falha que devem ser detectados e quais componentes do NFV-MANO são responsáveis por cada tipo de falha. Em geral, as falhas de *hardware* (*i.e.*, processamento, armazenamento e rede) são detectadas e corrigidas pelo VIM, podendo ser complementadas pelo VNFM e NFVO.

Já para as falhas de software (*i.e.*, a aplicação responsável por executar a função de rede), as próprias VNFs são responsáveis por detectar falhas do seu próprio contexto sem depender de nenhum componente do NFV-MANO (Schöller et al., 2015). Por fim, as falhas detectadas pelo VIM ou pela própria VNF em sua aplicação, devem ser reportadas ao NFV-MANO para que as ações correspondentes de recuperação sejam executadas.

A prevenção de falhas tem como objetivo evitar que novas falhas ocorram. A prevenção de falhas ocorre durante o próprio tempo de execução, aprendendo com falhas anteriores e, ativando preventivamente, procedimentos para evitar que novas falhas ocorram (Han et al., 2017). Uma das maneiras de garantir a prevenção de falhas é através do isolamento, evitando que a falha de um componente se propague para os demais. No contexto de NFV, o isolamento pode evitar que a falha de uma VNF se propague para as demais VNFs da SFC.

Por fim, a etapa de recuperação consiste em retomar o estado do sistema exatamente a partir do mesmo ponto antes da falha. Para tanto, diversos procedimentos podem ser executados, incluindo a ativação de réplicas (*i.e.*, substituir uma VNF falha pela sua réplica), recriar VNFs falhas, reconfiguração de rede (*e.g.*, criação de novas conexões, atribuição de endereços IP), bem como a recuperação do estado interno das VNFs.

Além disso, é importante garantir que a recuperação seja feita de maneira automatizada, *i.e.*, sem a intervenção do administrador da rede, de forma a mascarar a falha e minimizar o tempo de interrupção no serviço. Para tanto, cinco propriedades são essenciais ao desenvolver os mecanismos de recuperação (Han et al., 2017). A primeira é a corretude do serviço, garantindo que o estado da VNF seja consistente com o estado antes da falha. A segunda propriedade é minimizar a sobrecarga imposta pela plataforma NFV durante a execução sem falhas das VNFs, garantindo, por exemplo, que a latência no processamento da VNF não comprometa o desempenho do serviço. A terceira propriedade é garantir o menor tempo possível na recuperação de uma VNF, minimizando o tempo de indisponibilidade do serviço. A quarta propriedade é projetar os mecanismos de recuperação de maneira genérica e com o mínimo de modificações para o código interno da VNF. Por fim, a quinta propriedade define que os mecanismos de recuperação não devem aumentar os custos operacionais e de gerenciamento.

Para evitar pontos únicos de falha (quarto requisito de alta disponibilidade definido anteriormente), a ETSI prevê diversos métodos de resiliência, descritos a seguir.

3.5.2 Métodos de Resiliência

Uma das maneiras de aumentar a resiliência dos serviços oferecidos e executados em uma plataforma NFV é através da redundância. No contexto de NFV, os métodos de redundância podem ser classificados como (Lac et al., 2017): (i) sem redundância; (ii) *Active-Standby* e; (iii) *Active-Active*. Estes métodos são descritos a seguir.

No método sem redundância, como o nome sugere, a VNF não possui nenhuma réplica associada. É importante notar que, apesar deste método ser desprovido de redundância, a VNF pode possuir um grau de resiliência, visto que neste caso em geral os procedimentos de recuperação criam um novo dispositivo virtual para substituir a VNF falha e importam o estado interno a partir de um componente externo. Esta é a configuração de redundância mais simples, possuindo o menor custo computacional em troca de um maior tempo de recuperação.

Já no método *Active-Standby*, cada VNF ativa possui uma ou mais réplicas, denominadas de *standby*. As réplicas em *standby* são dispositivos virtuais que já estão criados, porém, não estão ativos, não executando nenhum tipo de operação. O método *Active-Standby* pode ser configurado de três maneiras diferentes:

- *hot-standby*: a cada intervalo de tempo o estado interno da VNF ativa é transmitido para a réplica em *standby*. Durante a transmissão, a VNF ativa é bloqueada, garantindo que não exista perda de pacotes em caso de falha. Naturalmente, esta configuração possui maior impacto na latência do serviço.
- *warm-standby*: de maneira semelhante ao *hot-standby*, o estado interno da VNF ativa é transmitido para a réplica, porém sem bloquear a sua execução. Em casos de falha é possível que alguma informação seja perdida.
- *cold-standby*: nesta configuração a recuperação é somente iniciada quando uma falha na VNF ativa é detectada, possivelmente gerando grande perda de dados.

Por fim, no método *Active-Active* as VNFs possuem uma ou mais réplicas que estão em estado ativo, oferecendo o mesmo serviço simultaneamente. Para tanto, o estado interno da VNF ativa é constantemente replicado para as suas réplicas.

Para os métodos *Active-Standby* e *Active-Active*, a forma como as VNFs e réplicas são configuradas pode ser classificada nos seguintes cenários:

- $1 + 1$: para cada VNF ativa existe uma réplica ativa ou em *standby*.
- $N + 1$: para um grupo de N VNFs ativas, existe uma única réplica ativa ou em *standby*.
- $N + M$: também conhecido como *cluster redundancy*, é a generalização desta configuração, permitindo que cada VNF ativa (de um total de N VNFs) tenham uma ou mais réplicas ativas, ou em *standby* (totalizando M réplicas).

Considerando os requisitos de resiliência especificados pela ETSI, o próximo capítulo descreve uma estratégia para garantir a alta disponibilidade tanto de VNFs quanto SFCs.

4 UMA ARQUITETURA DE ALTA DISPONIBILIDADE PARA SERVIÇOS VIRTUALIZADOS

Apesar das diversas vantagens oferecidas pela tecnologia NFV, a confiabilidade dos serviços virtualizados deve ser considerada, de forma a garantir que tais serviços tenham níveis de disponibilidade adequados. Em particular, é essencial que um ambiente NFV forneça mecanismos que garantam a execução correta e contínua de seus serviços. Este capítulo explora a arquitetura NHAM (NFV *High Availability Module*), projetada como um módulo totalmente compatível com a arquitetura de referência NFV-MANO.

O restante deste capítulo está organizado da seguinte forma. A Seção 4.1 descreve os desafios de disponibilidade e confiabilidade que podem surgir em um ambiente virtualizado. Em seguida, a Seção 4.2 apresenta a arquitetura NHAM, descrevendo em detalhes as estratégias para tolerar falhas de VNFs individuais. Já a Seção 4.3 descreve os procedimentos utilizados pelo NHAM para garantir também a tolerância a falhas de SFCs. Em seguida, a Seção 4.4 apresenta a implementação e resultados experimentais. Por fim, a Seção 4.5 apresenta os trabalhos relacionados enquanto que a Seção 4.6 conclui o capítulo.

4.1 EXPLORANDO OS DESAFIOS DE CONFIABILIDADE EM AMBIENTES NFV

As tecnologias de virtualização representam uma estratégia concreta para permitir a evolução da rede para múltiplas direções. Em particular, o paradigma NFV explora a programabilidade da rede, melhorando consequentemente a sua flexibilidade e gerenciamento. No entanto, é inquestionável que serviços de rede executados de maneira virtualizada são mais propensos a falhas em comparação com as alternativas tradicionais disponibilizada em hardware especializado (Han et al., 2017, 2015; Gunawi et al., 2016). A transição dos dispositivos de hardware para plataformas virtualizadas introduz diversos desafios relacionados à confiabilidade dos sistemas (Sharma et al., 2020; Li et al., 2020; Cotroneo et al., 2014b; Mijumbi et al., 2016). Em particular, a complexidade para integrar diferentes softwares em múltiplas camadas, os desafios de interoperabilidade entre componentes de hardware e software fornecidos por diferentes desenvolvedores e a inexperiência na operação de ambientes virtualizados de rede são alguns dos fatores que contribuem para uma expectativa de aumento de riscos em termos da confiabilidade das redes baseadas em NFV (Schöller et al., 2015).

Por outro lado, funções de rede implementadas em dispositivos especializados de hardware (*i.e.*, middleboxes) são normalmente projetadas para oferecer alta disponibilidade, atingindo o nível exigido pelos sistemas comerciais de telecomunicações: *five nines*, correspondendo a 99,999% do tempo em estado correto (Gray e Siewiorek, 1991) – aproximadamente 5 minutos de inatividade por ano. Dessa forma, torna-se essencial prover mecanismos que minimizem o tempo de inatividade dos serviços virtualizados na presença de falhas. Em especial, a ETSI definiu diversos requisitos de resiliência para serviços executados em ambientes NFV (Lac et al., 2017; Han et al., 2017; Schöller et al., 2015; Nakamura et al., 2016). Entretanto, a disponibilidade e continuidade do serviço tem sido negligenciados pela maioria dos desenvolvedores de VNFs e plataformas NFV (Han et al., 2017).

Diversas propostas já foram desenvolvidas para prover alta disponibilidade de funções e serviços de rede virtualizados. Tais propostas incluem soluções de posicionamento de VNFs (*VNF placement*) (Fan et al., 2018, 2017; Kong et al., 2017b), tolerância a falhas em outros componentes, tais como na própria NFVI (Herker et al., 2015), no NFVO (Gonzalez et al., 2018)

e no controlador SDN (*Software-Defined Network*) (Al-Rubaye et al., 2019), além de módulos e *frameworks* para prover resiliência e gerenciamento de falhas (Ghaznavi et al., 2020; Kulkarni et al., 2018; Khalid e Akella, 2019; Kablan et al., 2017; Li et al., 2015).

Entretanto, tais soluções possuem diversas limitações, como a falta de generalidade (*i.e.*, soluções que exigem tecnologias específicas) ou exigem modificações no código interno da VNF. Outras soluções não englobam todas as funcionalidades necessárias para garantir a confiabilidade de ponta-a-ponta nos serviços virtualizados. O problema torna-se ainda mais complexo ao considerar que grande parte dos serviços de rede são *stateful* – o estado interno da função de rede muda de acordo com os pacotes processados e de acordo com o próprio fluxo de execução da função. Consequentemente, o problema não pode ser resolvido apenas através da utilização de réplicas redundantes nem simplesmente recriando as funções que falharam. Dessa forma, a recuperação de um serviço *stateful* só é concluída se a restauração do seu estado antes da falha ocorrer. Por fim, as soluções não são totalmente compatíveis com a arquitetura de referência NFV-MANO, dificultando a sua integração com as plataformas NFV existentes.

Neste contexto, este capítulo propõe uma arquitetura de alta disponibilidade para serviços baseados em NFV, incluindo tanto VNFs quanto SFCs *stateful* e *stateless*. A arquitetura, denominada NHAM, é integrada como um módulo da arquitetura de referência NFV-MANO, projetada de maneira totalmente compatível com as especificações definidas pela ETSI. A estratégia adotada pelo NHAM é definida em nível de virtualização, permitindo que tanto VNFs isoladas quanto SFCs simplesmente herdem as propriedades de alta disponibilidade e resiliência de maneira transparente, *i.e.*, sem a necessidade de qualquer modificação do serviço.

O NHAM realiza o gerenciamento de falhas, além de gerenciar o estado interno das VNFs, oferecendo diversos mecanismos para garantir a alta disponibilidade dos serviços. Para o gerenciamento de falhas, o NHAM possui componentes responsáveis por detectar e recuperar VNFs. Para o gerenciamento do estado das VNFs, um componente é proposto para monitorar e controlar o estado interno de cada VNF com técnicas baseadas em *Checkpoint/Restore*. Dessa maneira, o NHAM garante que as VNFs são recuperadas de forma a também preservar o seu estado interno antes da falha. O NHAM oferece uma escolha de quatro mecanismos de resiliência diferentes, utilizados para configurar e atualizar as réplicas das VNFs. Tais mecanismos variam em termos de utilização computacional e tempo de recuperação, permitindo a recuperação de falhas de diferentes tipos de VNFs com diferentes requisitos de disponibilidade.

O NHAM foi projetado de maneira agnóstica em relação à tecnologia NFV: qualquer serviço baseado em NFV torna-se altamente disponível sem a necessidade do desenvolvedor/usuário realizar modificações em seu código-fonte. Como as VNFs executam em ambientes virtualizados, copiar o estado do dispositivo virtual torna-se uma estratégia concreta e atrativa, uma vez que esta abordagem não exige modificações no código-fonte da VNF. A viabilização desta estratégia ocorre através da realização de *checkpoints* diretamente nos recursos virtuais em que as VNFs estão executando.

Por fim, o NHAM garante não somente a alta disponibilidade de VNFs individuais como também de SFCs, tornando-se uma solução que garante a resiliência de serviços virtualizados. A estratégia proposta é baseada na junção de *checkpoints* com o gerenciamento de *buffers*, permitindo a sincronização do tráfego processado por cada VNF de acordo com os *checkpoints* correspondentes. Consequentemente, o NHAM garante a recuperação completa e correta de ponta-a-ponta dos serviços de rede, tolerando múltiplas falhas simultâneas, além de evitar a perda e a duplicação de pacotes.

A próxima seção apresenta a arquitetura NHAM, descrevendo em detalhes como VNFs isoladas tornam-se altamente disponíveis.

4.2 NHAM: NFV HIGH AVAILABILITY MODULE

A arquitetura NHAM define várias estratégias de resiliência e alta disponibilidade tanto para VNFs quanto para SFCs. As estratégias incluem a detecção e recuperação de falhas de serviços heterogêneos de diferentes desenvolvedores e fornecedores. Uma vez que a disponibilidade de um serviço aumenta na medida em que o tempo de recuperação diminui, é essencial detectar e reagir às falhas rapidamente, minimizando o tempo de inatividade.

O NHAM foi projetado como um módulo da arquitetura NFV-MANO. Para tanto, o NHAM se comunica com os demais módulos da arquitetura NFV-MANO, conforme mostra a Figura 4.1.

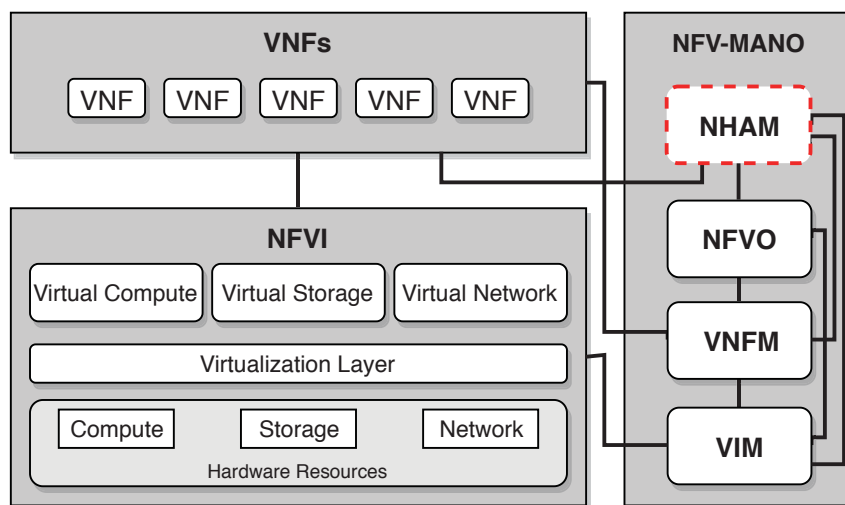


Figura 4.1: Arquitetura NFV-MANO e o NHAM.

Como visto no Capítulo 2, os módulos do NFV-MANO realizam, entre outras tarefas, operações relacionadas ao ciclo de vida dos serviços virtualizados, incluindo a instanciamento, remoção e atualização de VNFs. Portanto, o NHAM comunica-se com os componentes do NFV-MANO para realizar funções específicas para garantir a alta disponibilidade das VNFs. Por exemplo, durante os procedimentos de recuperação, o NHAM solicita a criação de novas VNFs através do VNFM e faz requisições ao VIM para o gerenciamento dos dispositivos virtuais das VNFs. Outro exemplo consiste na reconfiguração das SFCs através do NFVO.

As próximas subseções descrevem o modelo de sistema adotado pelo NHAM, além de detalhar os componentes internos e funcionalidades da arquitetura proposta.

4.2.1 Modelo de Sistema

O NHAM assume um modelo assíncrono e falhas por parada de uma ou múltiplas VNFs. O NHAM detecta e recupera falhas em um dispositivo virtual específico, *i.e.*, quando um *container* ou VM não está totalmente operacional. Tais erros podem ser causados, por exemplo, por erros no sistema operacional que hospeda a VNF, falta de recursos virtuais, interface de rede indisponível, entre outros.

Dessa forma, as situações a seguir não são tratadas pelo NHAM e estão fora do escopo deste trabalho: falhas no enlace de comunicação entre VNFs de uma SFC ou entre o NHAM e NFVI; falhas no nodo que hospeda as VNFs (*i.e.*, NFVI); falhas nos demais componentes do NFV-MANO e; falhas de hardware.

O NHAM considera e trata a falhas simultâneas de múltiplas VNFs, independente de estarem ou não em um mesmo nodo – o NHAM identifica individualmente cada falha e o local

onde elas ocorreram. Por fim, o NHAM assume que os seus próprios componentes não falham, uma vez que os componentes do NFV-MANO (e consequentemente o NHAM) são geralmente executados diretamente em hardware físico, sem virtualização.

4.2.2 A Arquitetura NHAM

A arquitetura NHAM consiste de dois componentes principais, conforme ilustra a Figura 4.2: o Sistema de Gerenciamento de Falhas (*Fault Management System* - FMS) e o Gerenciador de Estados de VNFs (*VNF State Manager* - VSM), ambos descritos a seguir.

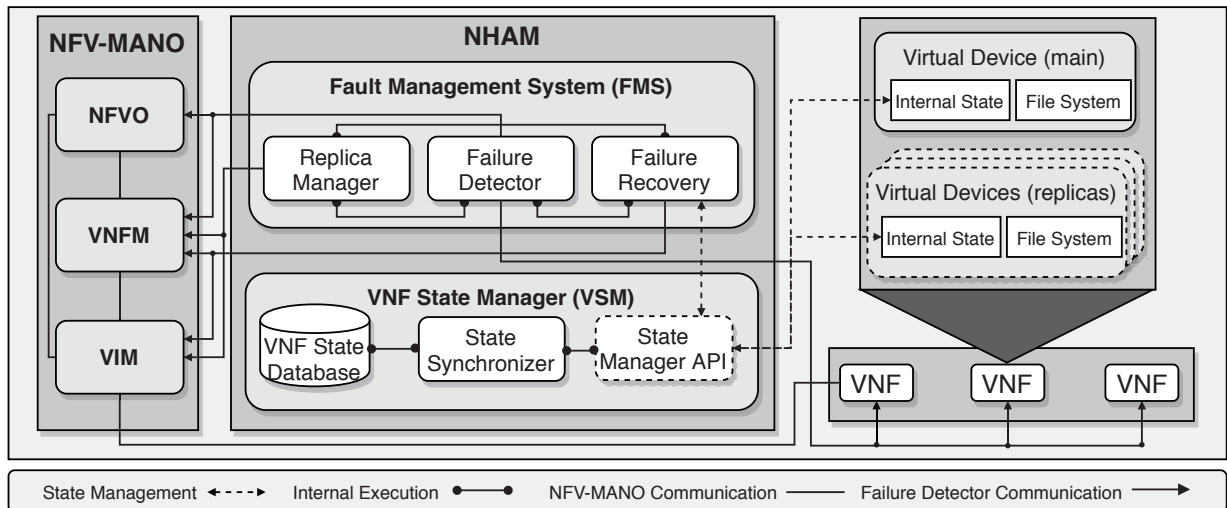


Figura 4.2: A arquitetura NHAM.

Uma das principais responsabilidades do componente FMS é a detecção de falhas. No contexto de NFV, a detecção de falhas consiste no monitoramento contínuo de cada instância de VNF, identificando possíveis falhas (Turchetti et al., 2015). O NHAM possui um módulo específico para a detecção de falhas, denominado de *Failure Detector* (FD). O FD consiste de duas estratégias. A primeira estratégia é baseada em *polling* – mensagens são enviadas periodicamente para as VNFs monitoradas. Para que a VNF seja considerada correta, as mensagens correspondentes de confirmação devem ser enviadas e recebidas pelo NHAM antes de um intervalo de *timeout*, calculado de maneira adaptativa.

Além das mensagens de *polling*, o FD também inspeciona o dispositivo virtual que está hospedando a VNF (*i.e.*, *container* ou VM) – funcionalidade presente em grande parte dos *hypervisors*. Em geral, o resultado desta inspeção retorna um dos seguintes resultados: *active*, *exited*, *stopped* ou *error*. Para qualquer resultado diferente de *active*, o FD considera a VNF como suspeita de falha. Imediatamente após uma VNF ser identificada como suspeita de falha, o seu identificador é adicionado pelo FD para uma lista de suspeitos e uma mensagem de notificação é enviada ao NHAM para realizar os procedimentos de recuperação.

Após a detecção de uma falha, o NHAM precisa obter o local específico em que a VNF falha estava executando. Para tanto, o NHAM envia uma requisição ao VNFM que retorna o servidor que está hospedando a VNF bem como o seu respectivo endereço IP. Após terminar a localização, o NHAM envia uma mensagem para o VNFM, notificando que a VNF deve estar em processo de recuperação. Os processos de recuperação diferem para cada método de resiliência empregado pelo NHAM.

Além da detecção de falhas, o FMS possui um subcomponente denominado de *Replica Manager*, cuja responsabilidade é permitir ao cliente/usuário/operador da rede uma escolha

de diferentes mecanismos de resiliência. Tais mecanismos de resiliência são utilizados para determinar qual o tipo de replicação será adotada por cada VNF instanciada na rede. Para cada instância, quatro diferentes mecanismos de resiliência podem ser especificados. A escolha de qual mecanismo utilizar é feita de acordo com os requisitos específicos de disponibilidade de cada VNF. Os quatro mecanismos de resiliência e suas diferenças serão descritos em detalhes nas Seções 4.2.4 e 4.2.5.

Por fim, o FMS também é responsável pela recuperação de VNFs, realizada pelo módulo *Failure Recovery*. Em particular, é essencial preservar o estado interno de uma VNF após a sua recuperação, de forma a garantir a recuperação completa, correta e consistente de VNFs *stateful*. Para tanto, o NHAM utiliza um gerenciador de estados, representado pelo componente VSM. O VSM consiste de um subcomponente específico para sincronizar o estado das VNFs (*State Synchronizer*), além de contar com uma API que fornece operações para manipular de maneira simplificada o estado das VNFs, permitindo a importação e exportação de estados. O VSM é descrito em detalhes a seguir.

4.2.3 Gerenciando o Estado de VNFs *Stateful*

De maneira a permitir a recuperação de VNFs *stateful*, o NHAM define um gerenciador de estados (VSM) baseado em *Checkpoint/Restore*. Periodicamente, *checkpoints* do estado interno são capturados e salvos em memória não volátil, contendo uma representação do estado do sistema (*e.g.*, descritores de arquivos, mapeamento de memória). Quando uma falha é detectada, o sistema pode ser recuperado (*i.e.*, *restore*) para o *checkpoint* mais recente.

No contexto de NFV, o estado de uma VNF pode ser classificado como externo ou interno (Nakamura et al., 2016; Han et al., 2017; Kablan et al., 2017; Rajagopalan et al., 2013b). No estado externo, também referenciado como estado estático, são armazenadas informações que não variam com o tempo ou que variam ocasionalmente. Regras de um *firewall*, regras de um IDS e o mapeamento de portas de um NAT são exemplos de informações armazenadas no estado externo de uma função de rede. Em casos de falha, o estado externo pode ser facilmente recuperado após a reinicialização da VNF.

O estado interno, por sua vez, também conhecido como estado dinâmico, consiste de informações que são atualizadas de acordo com o processamento de pacotes e a execução de processos. O mapeamento de memória, conexões TCP e conteúdos de cache são exemplos de informações que pertencem ao estado interno de uma função de rede. O principal desafio no gerenciamento de estados é preservar e garantir a consistência do estado interno das VNFs em casos de falha e recuperação.

O principal componente do VSM é o *State Synchronizer*, responsável por capturar as informações do estado das VNFs e as armazenar em *checkpoints*. Em particular, o *checkpoint* consiste das seguintes informações: árvore de processos, PIDs (*Process Identifier*), *threads*, mapeamento de memória, arquivos abertos, conexões de rede (TCP, UDP) e chamadas ao *kernel*.

Para cada VNF, o *State Synchronizer* cria um agente que coleta periodicamente o estado interno da VNF através de *checkpoints*. Uma vez que cada instância de VNF possui o seu agente correspondente, é possível detectar e iniciar procedimentos de recuperação em paralelo, otimizando cenários em que múltiplas falhas ocorrem simultaneamente. O intervalo de *checkpoint* é customizável e pode ser especificado de maneira específica para cada VNF.

O NHAM implementa uma API, denominada de *State Manager API*, para gerenciar o estado interno das VNFs de maneira transparente, sem exigir nenhuma modificação no código-fonte da VNF. O objetivo desta API é utilizar técnicas e ferramentas que inspecionem diretamente a memória e o estado da CPU para obter informações que podem ser posteriormente utilizadas para reconstruir o estado da VNF naquele momento. Esta API deve possuir ao menos

duas primitivas principais. A primeira primitiva deve capturar as informações necessárias (*i.e.*, exportar o estado interno da VNF), enquanto que a segunda primitiva deve restituir as informações (*i.e.*, importar o estado em uma VNF).

É importante observar que esta API pode ser implementada de diversas formas, de acordo com os requisitos do sistema. Em particular, o NHAM emprega ferramentas de *Checkpoint/Restore* em *containers* para estas funcionalidades. No entanto, outros tipos de ferramentas podem ser utilizadas, como *checkpoints* para VMs, técnicas de *live migration*, ou até mesmo implementações customizadas e específicas para um determinado tipo de VNF. O NHAM implementa a *State Manager API*, que reside no componente VSM. Em especial, o *State Synchronizer* utiliza esta API para capturar periodicamente *checkpoints* das VNFs e importar os *checkpoints* em situações de falha. As duas primitivas principais da *State Manager API* são descritas a seguir.

export_vnf_state. Esta operação é utilizada para realizar o *checkpoint* de uma VNF específica. Primeiramente, o dispositivo virtual deve ser momentaneamente parado para que o *checkpoint* possa ser gerado de maneira consistente. Após receber uma confirmação, o *State Synchronizer* gera um *timestamp* correspondente e o *checkpoint* é enviado para a base de dados (VNF State Database). A VNF State Database armazena os *checkpoints* de todas as VNFs instanciadas. Dependendo do método de resiliência adotado, o *checkpoint* pode também ser enviado diretamente para as réplicas. De forma a reduzir o impacto na sobrecarga causada pelas operações de sincronização, o *checkpoint* é feito de maneira assíncrona – a VNF continua a sua execução enquanto o *State Synchronizer* realiza o envio *checkpoint* para o destino correto.

import_vnf_state. Esta operação é utilizada para restaurar o estado de uma VNF com um *checkpoint*, *i.e.*, retomar a VNF para um estado prévio. Nesta operação, dois parâmetros são especificados: (i) *vnf_id* especifica qual VNF receberá o estado atualizado e; (ii) *checkpoint_id*, que identifica o *checkpoint* que deverá ser inserido na VNF em questão. Assim como na operação *export_vnf_state*, o primeiro passo é parar momentaneamente a VNF que receberá o *checkpoint*. Em seguida, o *checkpoint* é inserido no dispositivo virtual e a VNF é atualizada. Após completar a importação, o *State Synchronizer* retorna uma mensagem indicando se a operação foi realizada com sucesso.

É importante notar que apesar de o gerenciamento de estado ser obrigatório para a recuperação de VNFs *stateful*, ele não é necessário para funções de rede *stateless*. Portanto, o NHAM permite o usuário decidir se a VNF terá o seu estado interno preservado.

4.2.4 Mecanismos de Resiliência e Recuperação de Falhas

Escolher uma estratégia que garanta a alta disponibilidade de VNFs depende dos requisitos de resiliência específicos de cada função de rede (Schöller et al., 2015). Por exemplo, funções de rede que processam tráfego em tempo real possuem requisitos mais restritos do que funções que processam tráfego de baixa prioridade. Portanto, a plataforma NFV deve suportar diferentes estratégias com diferentes propriedades, fazendo bom uso dos recursos do sistema.

Conforme descrito no Capítulo 3.5, os métodos de redundância definidos pela ETSI podem ser classificados como sem redundância, *Active-Standby* ou *Active-Active*. O NHAM oferece quatro diferentes mecanismos de resiliência baseados nestes métodos de redundância, que variam em termos de custo computacional e tempo de recuperação, adaptando-se a diferentes circunstâncias e tipos de VNFs. Os procedimentos de cada abordagem são descritos a seguir.

4.2.4.1 No Redundancy (0R)

Como o nome sugere, o mecanismo 0R não aplica nenhum tipo de redundância, conforme mostra a Figura 4.3. Neste caso, o *State Synchronizer* somente exporta periodicamente os *checkpoints* da VNF para a *VNF State Database*. Esta abordagem é a mais simples e com menor utilização de recursos computacionais, uma vez que não utiliza nenhum recurso virtual para manter nenhuma réplica.

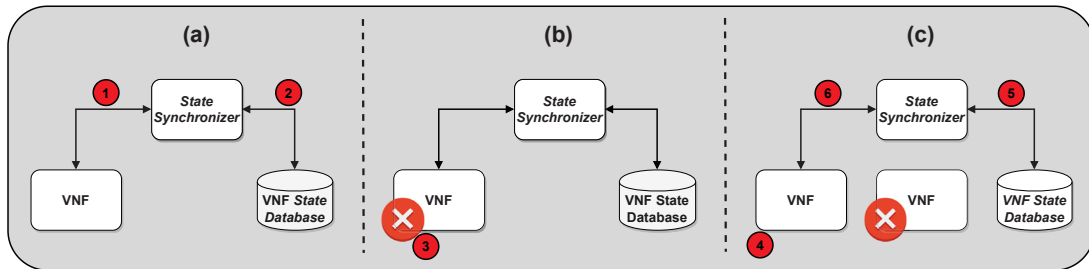


Figura 4.3: Funcionamento do mecanismo 0R.

Por outro lado, em casos de falha, o tempo de recuperação será maior comparado aos demais métodos. Uma vez que o mecanismo 0R não utiliza nenhuma réplica, o primeiro passo no processo de recuperação consiste na criação de uma nova VNF, de forma a substituir aquela que falhou. Em seguida, o NHAM atualiza o estado interno da nova VNF. Para tanto, o *State Synchronizer* importa o *checkpoint* mais recente da VNF falha na nova VNF. Uma vez que o processo de recuperação é finalizado, o processo de reconfiguração inicia. O primeiro passo é obter as informações da VNF recém criada, tais como endereço IP e outros identificadores. O NHAM então envia essas informações atualizadas para o NFV-MANO, notificando que uma VNF foi substituída e que as informações relacionadas à VNF podem ser atualizadas.

4.2.4.2 Primary Replica Active-Standby (1R-AS)

O Mecanismo 1R-AS utiliza o método *Active-Standby* (*warm-standby*) com uma configuração 1 + 1. Dessa forma, a VNF inicia com uma réplica que está instanciada, mas permanece em modo *standby*. A Figura 4.4 ilustra este comportamento. Assim como no mecanismo 0R, o *State Synchronizer* exporta periodicamente o *checkpoint* da VNF para a *VNF State Database*. Esta opção é mais custosa que o método 0R, pois consome recursos da infraestrutura virtualizada para manter a réplica (armazenamento e rede), porém possui menor tempo de recuperação.

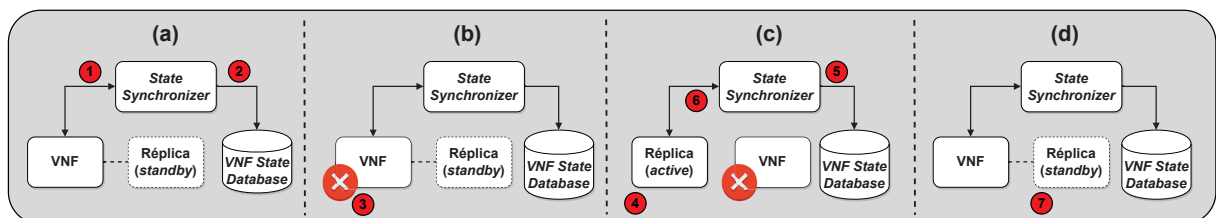


Figura 4.4: Funcionamento do mecanismo 1R-AS.

Em caso de falha, a réplica já está criada e o *State Synchronizer* precisa apenas importar o *checkpoint* mais recente da VNF falha, reduzindo o tempo de instanciação de uma nova réplica. Para a reconfiguração, a réplica se torna a VNF principal e o NHAM envia uma requisição para o NFV-MANO atualizando as informações necessárias. Por fim, uma nova réplica é criada em modo *standby* para substituir a réplica utilizada no procedimento de recuperação.

4.2.4.3 Primary Replica Active-Active (1R-AA)

O mecanismo 1R-AA é baseado no método *Active-Active* com configuração 1 + 1, no qual a VNF possui uma réplica ativa. Como pode ser observado na Figura 4.5, o *State Synchronizer* atualiza periodicamente a réplica com os *checkpoints* da VNF. Portanto, a réplica recebe constantemente atualizações que refletem o estado da VNF primária. Naturalmente, a estratégia 1R-AA é mais custosa do que a 1R-AS, no entanto, o tempo de recuperação é ainda menor.

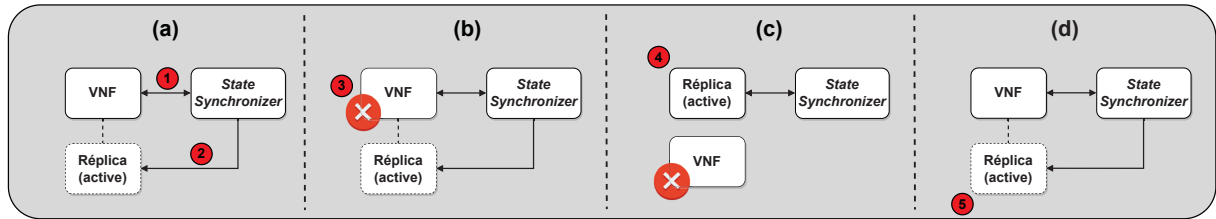


Figura 4.5: Funcionamento do mecanismo 1R-AA.

Na ocorrência de uma falha, nenhum procedimento de recuperação é necessário, visto que a réplica está sempre atualizada com o *checkpoint* mais recente da VNF. Neste caso, somente o procedimento de reconfiguração é executado. Para tanto, o NHAM apenas indica que a réplica é agora a VNF primária, e uma nova réplica é criada para substituir aquela que se tornou a primária.

4.2.4.4 Multiple Replicas Active-Active (MR-AA)

O mecanismo MR-AA é a generalização do mecanismo 1R-AA, baseando-se no método *Active-Active* com configuração 1 + M . Nesta estratégia, ilustrada na Figura 4.6, uma VNF pode ser vista como um grupo de M réplicas, constantemente sincronizadas de acordo com a VNF primária. O *State Synchronizer* é responsável por manter o estado das M réplicas consistente. Além disso, qualquer réplica deste grupo pode ser acessada para utilizar o serviço. O MR-AA é o mecanismo mais custoso de todos, uma vez que exige a sincronização de M réplicas, porém é a que possui o menor tempo de inatividade em casos de falha.

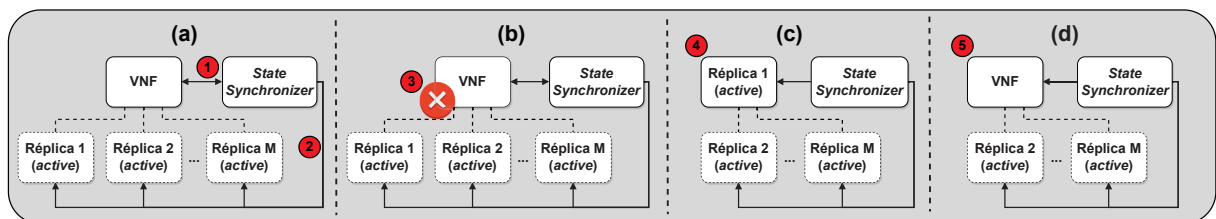


Figura 4.6: Funcionamento do mecanismo MR-AA.

Em casos de falha, o mecanismo também não exige nenhum procedimento de recuperação, exigindo somente que uma nova VNF do grupo se torne a VNF primária, em caso de falha desta. O NHAM permite, no entanto, que o usuário defina um número mínimo e máximo de réplicas para o grupo. Se a quantidade de réplicas do grupo exceder o limite mínimo, o NHAM cria novas réplicas (até o número máximo especificado).

4.2.5 Comparação dos Mecanismos de Resiliência

Os diferentes métodos de resiliência podem ser comparados entre si com base na relação entre a utilização de recursos computacionais (*e.g.*, uso de memória e CPU) e o tempo de recuperação. A Tabela 4.1 apresenta uma comparação entre as diferentes estratégias de resiliência.

Tabela 4.1: Comparação entre os diferentes tipos de mecanismos de resiliência.

Mecanismo de Resiliência	Método	# Réplicas	Base de Dados	Tempo de Recuperação	Utilização de Recursos
0R	Sem redundância	0	Sim	Mais alto	Mais baixo
1R-AS	<i>Active-Standby</i>	1	Sim	Moderado	Baixo
1R-AA	<i>Active-Active</i>	1	Não	Baixo	Alto
MR-AA	<i>Active-Active</i>	M	Não	Mais baixo	Mais alto

A estratégia 0R possui o menor custo computacional apesar de apresentar o maior tempo de recuperação. Esse mecanismo é ideal para VNFs que executam funções de rede de baixa prioridade e que podem tolerar um maior tempo de inatividade. Já a abordagem 1R-AS possui menor tempo de recuperação com relação ao 0R, uma vez que já possui uma réplica em *standby*. Essa estratégia é ideal para VNFs que possuem alta capacidade de disco, já que a réplica em *standby* não consome recursos de computação ou de rede.

Já as estratégias 1R-AA e MR-AA possuem o menor tempo de recuperação, pois o estado das réplicas é constantemente atualizado, ideais para VNFs que exigem altos níveis de disponibilidade. Por outro lado, elas possuem maior utilização de recursos computacionais.

De todos os métodos, o MR-AA é o mais complexo, uma vez que mantém um grupo de réplicas sincronizadas. Dessa forma, o *State Synchronizer* é utilizado como componente centralizado da arquitetura responsável por garantir que todo o grupo de réplicas seja atualizado de maneira correta e consistente. Para tanto, o *State Synchronizer* aguarda uma confirmação de que cada réplica recebeu o estado atualizado. Só após receber M confirmações (de um total de M réplicas), o *State Synchronizer* realiza a próxima sincronização. Para não aguardar indefinidamente por uma resposta de uma réplica que possivelmente falhou, o *State Synchronizer* faz um monitoramento baseado em *timeouts*.

Além disso, o NHAM garante a consistência dos estados das réplicas em outras duas situações específicas: (i) quando uma VNF é suspeita incorretamente e; (ii) quando uma VNF falha enquanto o estado ainda está em atualização. O primeiro caso é o mais simples, uma vez que se uma VNF é suspeita incorretamente, quando o engano é corrigido o NHAM simplesmente executa o procedimento de recuperação como se a falha realmente ocorreu. Para o segundo caso, considere que uma VNF falhou enquanto o *State Synchronizer* ainda está atualizando o grupo de réplicas. Neste caso, para evitar inconsistências, o *State Synchronizer* interrompe o procedimento e executa um *rollback* em todas as réplicas do grupo para o seu estado antes da atualização. A VNF falha é então removida do grupo e apenas na atualização seguinte as réplicas corretas recebem o *checkpoint* atualizado.

4.3 SERVIÇOS DE REDE ALTAMENTE DISPONÍVEIS

A Seção 4.2 descreveu a arquitetura NHAM e a estratégia empregada para garantir a alta disponibilidade de VNFs individuais. Esta seção detalha as estratégias adotadas pelo NHAM para a alta disponibilidade de SFCs.

O NHAM adota uma estratégia de alta disponibilidade para serviços virtualizados que depende dos componentes da arquitetura interna de SFCs proposta pela IETF, descrita na Seção 2.3. É importante notar que a maioria das estratégias para fornecer alta disponibilidade para SFCs é totalmente compatível com a arquitetura IETF. O NHAM utiliza componentes como o SFF e o SFP para permitir que SFCs *stateful* continuem operacionais mesmo após a falha de múltiplas VNFs.

4.3.1 Gerenciamento de *Buffers* e Premissas

O NHAM adota uma estratégia de alta disponibilidade que garante a recuperação de ponta-a-ponta de SFCs *stateful* – SFCs compostas por uma ou mais VNFs *stateful* – após um número arbitrário de VNFs falhar ao longo da cadeia. Na maior parte das implementações de SFC, cada VNF da cadeia é precedida por um *buffer*, utilizado pelo SFF para armazenar pacotes, antes de serem entregues para a VNF. Na estratégia adotada pelo NHAM, ao invés de empregar um único *buffer*, dois *buffers* são empregados para cada VNF da SFC, conforme mostra a Figura 4.7. Esses *buffers* são denominados de *buffer_rx* e *buffer_tx*. O *buffer_rx* precede a VNF e recebe o tráfego a ser entregue para a VNF. Dessa forma, o *buffer_rx* armazena os pacotes que ainda não foram processados pela VNF. Já o *buffer_tx* sucede a VNF, recebendo o tráfego já processado na saída. Portanto, o *buffer_tx* armazena o tráfego que já foi processado pela VNF.

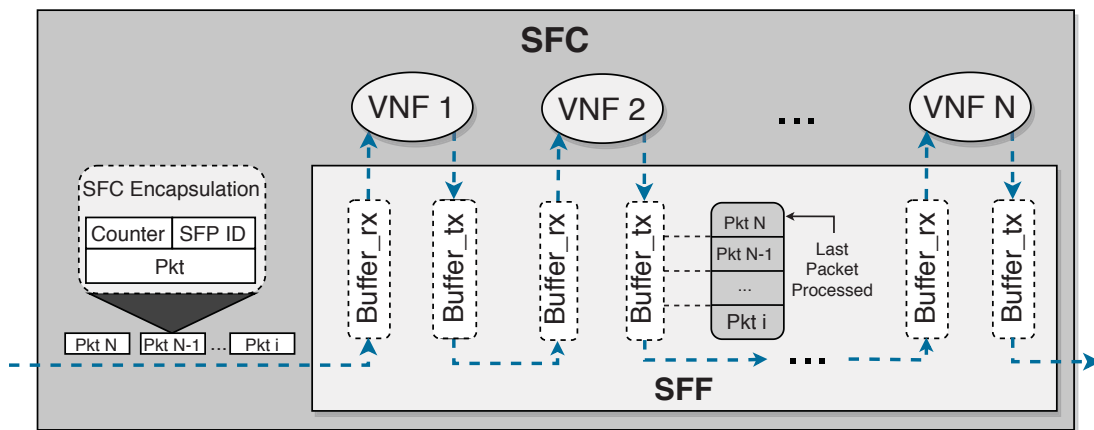


Figura 4.7: A arquitetura para SFCs altamente disponíveis.

O fluxo de dados inicia quando o *buffer_rx* da primeira VNF da SFC recebe do SFF os pacotes que serão processados pela VNF. Em seguida, o SFF encaminha os pacotes do *buffer_rx* para a VNF. Após processar o tráfego de acordo com a função de rede, a VNF envia os pacotes para o *buffer_tx*. O SFF atua novamente movendo pacotes do *buffer_tx* de uma VNF para o *buffer_rx* da próxima VNF da SFC. Este processo é repetido para as demais VNFs da SFC. Após a última VNF processar o tráfego e inserir os pacotes no último *buffer_tx*, o SFF entrega adequadamente esse tráfego ao destino.

O NHAM assume que os *buffers* e o SFF não falham. Esta é uma premissa realista, uma vez que para oferecer suporte a SFCs altamente disponíveis, o ambiente no qual as SFCs são executadas deve ter sido projetado para ser tolerante a falhas (o Capítulo 5 detalha uma estratégia neste contexto). Além disso, as VNFs não são compartilhadas entre múltiplas SFCs, *i.e.*, cada VNF é utilizada somente em uma única SFC. Portanto, é recomendável executar as VNFs e os *buffers* em hardwares fisicamente separados. Por fim, a estratégia de recuperação para SFCs assume que todas as VNFs da SFC processam tráfego considerando a ordem FIFO (*First-In First-Out*). Assim, sempre que dois pacotes são processados e enviados em uma determinada

ordem para a VNF, se eles não forem descartados pela VNF, eles são enviados preservando essa ordem.

A estratégia *Hold/Release* é proposta a seguir para garantir a recuperação consistente de uma SFC após uma ou mais falhas. A estratégia *Hold/Release* é baseada em uma combinação de *Checkpoint/Restore* de VNFs com o gerenciamento de *buffers* de uma SFC.

4.3.2 A Estratégia SFC *Hold/Release*

A recuperação de uma SFC *stateful* consiste de duas partes principais: (i) a recuperação individual de cada VNF que falhou, incluindo a restauração do estado interno no caso de VNFs *stateful* (descrito na Seção 4.2.4) e; (ii) a retransmissão do tráfego perdido durante a inatividade de uma VNF em decorrência de uma falha, a qual é feita através da estratégia *Hold/Release*, descrita a seguir.

Antes que o SFF insira cada pacote no primeiro *buffer_rx* da primeira VNF da SFC, cada pacote é encapsulado para que possa ser roteado ao longo do SFP. Além de transportar informações explícitas para identificar o SFP (Halpern e Pignataro, 2015), o NHAM inclui no encapsulamento do SFF um *timestamp*, implementado como um contador e que funciona como um identificador de pacote sequencial.

O NHAM verifica o estado das VNFs através do monitoramento contínuo de cada instância, identificando possíveis falhas. Caso alguma falha seja detectada, o NHAM envia imediatamente uma mensagem ao SFF correspondente, solicitando a alteração do estado da SFC para *recovering* (em recuperação). Quando uma SFC está neste estado, o tráfego não progride através do SFP até que as falhas sejam completamente recuperadas. Consequentemente, uma VNF em estado de recuperação não recebe pacotes do *buffer_rx* nem envia pacotes para o *buffer_tx*.

Como mencionado acima, o SFF armazena no *buffer_rx* os pacotes a serem processados pela VNF. Na estratégia *Hold/Release*, após os pacotes do *buffer_rx* serem entregues à VNF, eles não são imediatamente removidos do *buffer*, i.e., eles são temporariamente retidos (estágio *Hold*). A VNF então recebe e processa os pacotes e armazena o tráfego resultante no *buffer_tx*. Periodicamente, *checkpoints* das VNFs são realizados pelo NHAM. A ideia é manter o tráfego no *buffer_rx* até que um *checkpoint* da VNF seja realizado após ter processado uma sequência de pacotes. Se um *checkpoint* for executado após um determinado pacote ter sido processado pela VNF, dizemos que o *checkpoint* inclui esse pacote.

Durante a realização de um *checkpoint*, a VNF é momentaneamente pausada (i.e., nenhum tráfego é recebido ou enviado) e o SFF aguarda até que este procedimento seja finalizado. A partir deste momento, é possível concluir que o último pacote no *buffer_tx* foi processado pela VNF e incluído no *checkpoint*. Antes de liberar a VNF para retomar a sua execução, o SFF remove todos os pacotes até o último pacote do *buffer_rx* (*Release*). Caso a VNF falhe antes que o *checkpoint* seja executado, a VNF é revertida para o *checkpoint* anterior e os procedimentos de recuperação descritos na Seção 4.2.4 são realizados. Além disso, todos os pacotes que a VNF recebeu a partir do último *checkpoint* (que ainda estão no *buffer_rx*) devem ser enviados novamente e processados pela VNF. Este último passo é essencial para garantir a consistência do serviço, visto que cada VNF recebe e processa tráfego entre *checkpoints*.

Considere como exemplo que todos os pacotes até o pacote i foram processados por uma VNF quando um *checkpoint* é iniciado. Considere que o pacote $i + 1$ também foi enviado do *buffer_rx* para a VNF, mas não foi incluído no *checkpoint*. Quando o *checkpoint* é concluído, o SFF confirma que o último pacote que já estava no *buffer_tx* é o pacote i que, portanto, foi incluído no *checkpoint*. Neste momento, todos os pacotes até o i podem ser removidos do *buffer_rx*. Observe que o pacote $i + 1$ ainda não pode ser removido, pois ele não foi incluído no

checkpoint e pode ser necessário reenviá-lo e reprocessá-lo pela VNF novamente em casos de falha. O SFF também mantém o *timestamp* do último pacote incluído no último *checkpoint*, além do último pacote entregue à próxima VNF da SFC (último pacote armazenado no *buffer_tx*). Essa estratégia evita o envio de pacotes duplicados para a próxima VNF depois que uma VNF em recuperação reprocessa o tráfego para reconstruir seu estado.

A comunicação entre VNFs em uma SFC traz desafios adicionais para a gerência de pacotes. Considere, por exemplo, outra situação em que os pacotes i , $i + 1$ e $i + 2$ foram enviados do *buffer_rx* para a VNF₁. No entanto, a VNF₁ processou apenas até o pacote i , quando um *checkpoint* foi concluído. O SFF pode então remover todos os pacotes até o i do *buffer_rx*. Considere, entretanto, que após a conclusão do *checkpoint*, que a VNF₁ continua o seu processamento e envia os pacotes $i + 1$ e $i + 2$ para o *buffer_tx*, e que em seguida já são encaminhados para a VNF₂. O SFF mantém o registro do último pacote de cada VNF no *buffer_tx*, neste caso o pacote $i + 2$. Considere agora que a VNF₁ falha neste momento. Como os pacotes $i + 1$ e $i + 2$ não foram incluídos no último *checkpoint*, eles devem ser reprocessados após a recuperação da VNF₁. No entanto, eles já foram encaminhados para a VNF₂. Neste caso, o SFF evita pacotes duplicados encaminhando para a VNF₂ somente os pacotes com identificador maior que o último pacote registrado, *i.e.*, pacotes $i + 3$ em diante.

A recuperação da VNF ocorre de acordo com o seu respectivo mecanismo de resiliência, conforme descrito na Seção 4.2. Após a VNF retomar o seu funcionamento com o seu respectivo estado interno restaurado, o próximo passo é a retransmissão de todo o tráfego em *buffer_rx*, incluindo os pacotes que a VNF recebeu desde que o último *checkpoint* foi salvo.

A estratégia *Hold/Release* garante que a SFC se recupere independentemente do número e de quais VNFs falharam ao longo do SFP. Múltiplas VNFs podem inclusive falhar simultaneamente, por exemplo, devido a uma queda de energia. Observe que o *buffer_rx* de uma determinada VNF é somente esvaziado após um *checkpoint* ser salvo. Dessa forma, nem o tráfego processado entre *checkpoints* de cada VNF, nem o tráfego entre VNFs é perdido em decorrência de uma falha, garantindo a consistência da SFC de ponta-a-ponta.

4.4 IMPLEMENTAÇÃO E RESULTADOS EXPERIMENTAIS

Um protótipo da arquitetura NHAM foi implementado em uma plataforma NFV compatível com o modelo de referência NFV-MANO. A implementação da arquitetura NHAM é modular e integrada como um módulo da arquitetura de referência NFV-MANO. Dessa forma, o NHAM é facilmente integrado a qualquer plataforma NFV compatível com as especificações da ETSI (*e.g.*, OpenStack, OSM). O protótipo foi implementado em linguagem Python 3.

No protótipo do NHAM, as VNFs são baseadas em *containers* da plataforma Docker (Merkel, 2014), uma vez que eles possuem baixo tempo de inicialização, além de apresentar pouca utilização de memória. Para prover a funcionalidade de *Checkpoint/Restore* em *containers*, o Docker possui um *plugin* que utiliza o CRIU (*Checkpoint/Restore in Userspace*) – implementação da funcionalidade de *Checkpoint/Restore* para sistemas baseados em Linux (CRIU, 2019). Note que o NHAM não é limitado a virtualização baseada em *containers*. Para VNFs baseadas em máquina virtual, ferramentas como o *Micro-Checkpointing* (QEMU, 2022), e *live migration* (Clark et al., 2005), podem ser adotadas para salvar os *checkpoints* das VNFs.

Para implementar o componente *State Synchronizer*, a ferramenta CRIU foi utilizada para capturar o estado interno da VNF. Para cada VNF instanciada o *State Synchronizer* cria um respectivo agente. Esse agente é um processo responsável somente por salvar periodicamente o estado interno da VNF. Os procedimentos de recuperação são de responsabilidade do próprio *State Synchronizer*, visto que é ele quem se comunica com o FMS para obter atualizações das

falhas das VNFs. Os *checkpoints* incluem somente as informações mínimas necessárias para restaurar a VNF, incluindo a própria função de rede e alguns recursos relacionados, como mapeamentos de memória e árvore de processos (*process tree*).

Como visto no Capítulo 4, o NHAM permite que o próprio desenvolvedor defina diferentes níveis de resiliência para as VNFs. Essa definição é feita através do descritor VNFD. O exemplo a seguir mostra as principais configurações de um VNFD utilizado pelo NHAM:

```

1 topology_template:
2   node_templates:
3     capabilities:
4       nfv_compute:
5         properties:
6           mem_size: 512 MB
7           num_cpus: 4
8         properties:
9           type: container
10          image: ubuntu
11        resiliency:
12          num_backups: 3
13          cooldown: 250 ms
14          vnf_level:
15            type: MR-AA

```

Neste VNFD, o NFV-MANO instancia uma VNF em um *container* Ubuntu, com 4 CPUs e 512 MB de RAM. Para os métodos de resiliência, essa VNF possui três réplicas com redundância do tipo MR-AA, *i.e.*, além da instância da VNF principal, outras três instâncias irão atuar como réplicas ativas desta VNF, recebendo periodicamente o estado interno da VNF principal. Além disso, o intervalo de tempo entre *checkpoints* foi configurado para 250ms.

O protótipo do NHAM foi executado em um processador Intel Core i7 a 2.50 GHz com 8 núcleos, 16 GB de RAM, uma placa de rede Ethernet de 1 Gbps e Linux Ubuntu 20.04. Cada VNF consiste de um Ubuntu *server* com 256 MB de RAM e 1 CPU. O mecanismo MR-AA usa 3 réplicas por padrão. As VNFs utilizadas nos experimentos são encaminhadoras de pacotes (*packet forwarders*). Os *checkpoints* são executados a cada 500ms. O tamanho de cada *buffer* na SFC é de 64 KB.

Um conjunto de experimentos foi executado para avaliar o desempenho do NHAM e a disponibilidade atingida pelos serviços virtualizados com o suporte da solução proposta. O primeiro conjunto de experimentos avalia o impacto no tempo de recuperação conforme o tamanho da SFC aumenta e considerando os diferentes mecanismos de resiliência. O segundo conjunto de experimentos avalia a utilização de recursos computacionais, de acordo com cada estratégia de recuperação, comparando tanto o uso de CPU quanto de memória. O terceiro conjunto de experimentos avalia o impacto do NHAM na vazão, enquanto que o último experimento mede a disponibilidade dos serviços baseados em NFV suportados pelo NHAM. Cada experimento foi executado 10 vezes e as médias dos resultados são apresentadas com um intervalo de confiança de 95%.

4.4.1 Recuperação de Falhas

Uma vez que a redução no tempo de inatividade é particularmente importante para aumentar a disponibilidade dos serviços virtualizados, o primeiro conjunto de experimentos mede o tempo total de recuperação, desde a detecção da falha até a recuperação total da SFC. As falhas foram injetadas através de *scripts* que removem todas as conexões das VNFs, impedindo então que as mensagens de monitoramento sejam recebidas, gerando as suspeitas de falha.

O primeiro experimento, ilustrado na Figura 4.8, mede o tempo de recuperação após a falha de uma única VNF na SFC. Este experimento compara os quatro diferentes mecanismos de resiliência, enquanto que o número de VNFs na SFC aumenta de 1 até 8.

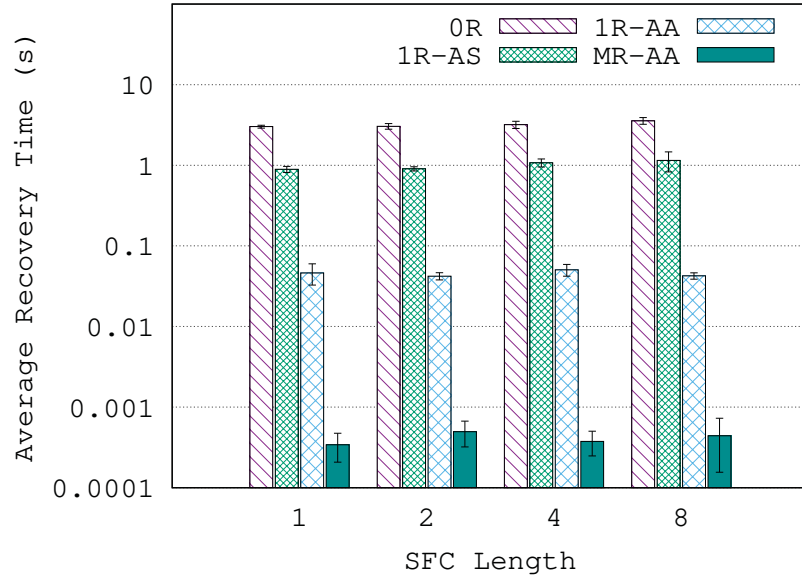


Figura 4.8: Tempo médio de recuperação após a falha de uma única VNF.

Como previsto, o mecanismo 0R apresenta o maior tempo de recuperação, atingindo até 3,6s de inatividade para uma SFC com 8 VNFs. Este é o tempo que leva para instanciar uma nova VNF (2,4s) e para o VSM restaurar o *checkpoint* mais recente. O mecanismo 1R-AS por sua vez apresenta resultados melhores, uma vez que a réplica já está instanciada em modo *standby*. Com o método *Active-Standby*, o NHAM somente importa um *checkpoint* para a réplica, reduzindo o tempo total de recuperação para 1,14s para uma SFC com 8 VNFs.

Mecanismos baseados no método *Active-Active* apresentam resultados ainda melhores. Em particular, o mecanismo 1R-AA atingiu uma média de 0,05s no tempo total de recuperação, uma vez que a réplica ativa já mantém o seu estado atualizado e o único procedimento a ser realizado é apontar a réplica como a VNF principal. Já o mecanismo MR-AA obteve os melhores resultados, devido ao seu grupo de réplicas sincronizadas. O seu tempo de recuperação é de apenas 0,0002s, o que corresponde à atualização do grupo de réplicas (*i.e.*, remoção da VNF falha). É importante ressaltar que, para todas as estratégias, o tempo de recuperação permanece inalterado conforme o tamanho da SFC aumenta.

O próximo experimento avalia o impacto no tempo de recuperação quando várias falhas ocorrem simultaneamente, por exemplo, devido a uma falha no enlace de comunicação ou a uma queda de energia. O experimento exibido na Figura 4.9 utiliza SFCs com 8 VNFs e o número de falhas por SFC varia de 1 (falha de uma única VNF) a 8 (falha de toda a SFC).

Como descrito na Seção 4.2.3, o NHAM detecta e recupera múltiplas VNFs com falha em paralelo. Neste sentido, foi medido o impacto no tempo de recuperação conforme a quantidade de falhas simultâneas aumenta. Para o mecanismo de resiliência 0R, o tempo de recuperação aumenta de 3,56s (uma falha) para 7,8s (8 falhas). Para o mecanismo 1R-AS, a diferença entre recuperar uma única falha e recuperar toda a SFC é menor: 1,14s para 2,52s.

Para os mecanismos 1R-AA e MR-AA a diferença é ainda menor, já que o tempo de recuperação de uma única falha é significativamente menor do que nas outras estratégias. Para a estratégia 1R-AA, o tempo de recuperação varia de 0,004s até 0,009s, enquanto que para o

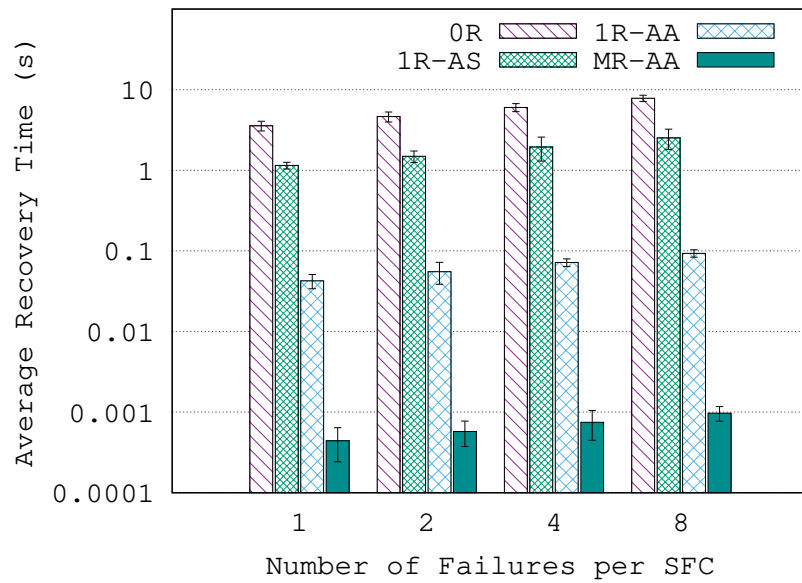


Figura 4.9: Tempo médio de recuperação de múltiplas falhas por SFC.

MR-AA o tempo de recuperação varia de 0,0004s até 0,0009s. A partir destes resultados, é possível concluir que todos os mecanismos de recuperação são escaláveis no que diz respeito ao número de falhas de VNFs por SFC.

4.4.2 Utilização de Recursos Computacionais

Os experimentos apresentados na seção anterior mostram como a estratégia proposta e os diferentes mecanismos diferem em termos de tempo de recuperação. No entanto, é necessário investigar quais os custos, em termos de memória e utilização da CPU, para monitorar, recuperar e sincronizar o estado interno das VNFs. As Figuras 4.10 e 4.11 mostram, respectivamente, os resultados de utilização de CPU e memória para cada um dos mecanismos de resiliência, conforme o tamanho da SFC varia de 1 a 8 VNFs.

O mecanismo 0R apresenta um maior tempo de recuperação em troca de um menor custo. O mecanismo 0R escala bem à medida que o número de VNFs aumenta: sua utilização de CPU permanece praticamente constante. Já para a utilização de memória, o aumento é proporcional ao número de VNFs, variando de 1,88% (1 VNF) a 13,04% (8 VNFs). Por outro lado, embora o mecanismo 1R-AS tenha um tempo de recuperação menor que o 0R, ele mantém os níveis de desempenho do 0R, tanto em termos de CPU quanto de memória.

Como esperado, mecanismos baseados no método *Active-Active* apresentam maior utilização de recursos, uma vez que estão constantemente sincronizando o estado interno de suas réplicas. O mecanismo 1R-AA apresenta utilização de CPU de 21% e uso de memória de 38% para sincronizar até 8 VNFs. Quanto ao mecanismo MR-AA, apesar de apresentar um uso de memória semelhante ao do 1R-AA, seu uso de CPU é maior: até 45% para uma SFC com 8 VNFs.

É importante observar que o uso de memória das abordagens baseadas no método *Active-Active* é significativamente maior do que as outras. Isso se deve ao fato de que o 1R-AA e o MR-AA realizam suas operações diretamente em memória, evitando sobrecargas de I/O em memória não-volátil.

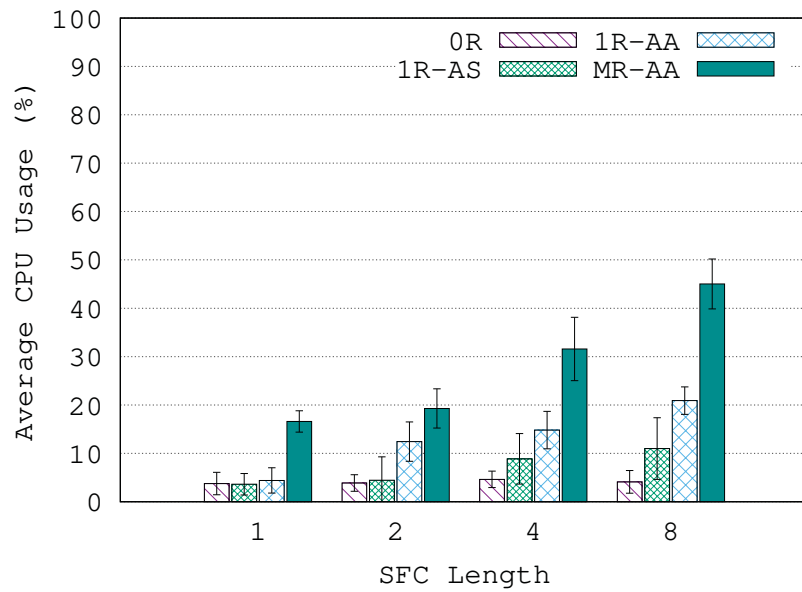


Figura 4.10: Utilização de CPU.

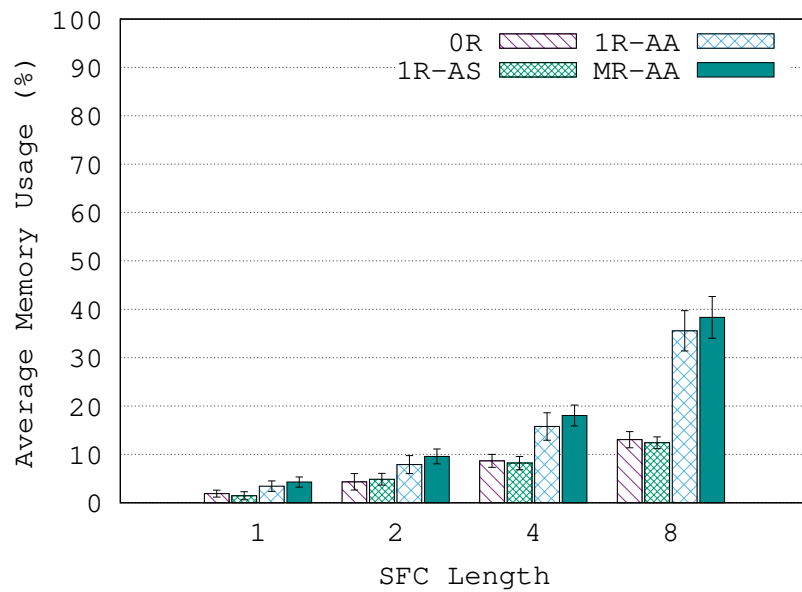


Figura 4.11: Utilização de memória.

4.4.3 Avaliando a Vazão das SFCs

O NHAM utiliza a estratégia *Hold/Release* para garantir a alta disponibilidade de serviços virtualizados. Os experimentos descritos nesta seção medem o impacto dessa estratégia com relação à vazão obtida pela SFC. O primeiro conjunto de experimentos avalia o impacto na vazão em dois cenários diferentes: o primeiro cenário é sem falhas enquanto que no segundo falhas ocorrem a cada 30 segundos. As SFCs nestes experimentos consistem de 4 VNFs.

No cenário sem falhas (Figura 4.12), cada mecanismo de resiliência é comparado ao *baseline* – uma SFC sem o suporte do NHAM. Os mecanismos 0R e 1R-AS apresentam vazão semelhante, já que ambos os mecanismos realizam *checkpoints* da mesma maneira. Esses mecanismos diminuem a taxa de transferência em aproximadamente 11,5%. Isso se deve ao fato

de que os *checkpoints* são obtidos, compactados e salvos em memória não-volátil em um banco de dados.

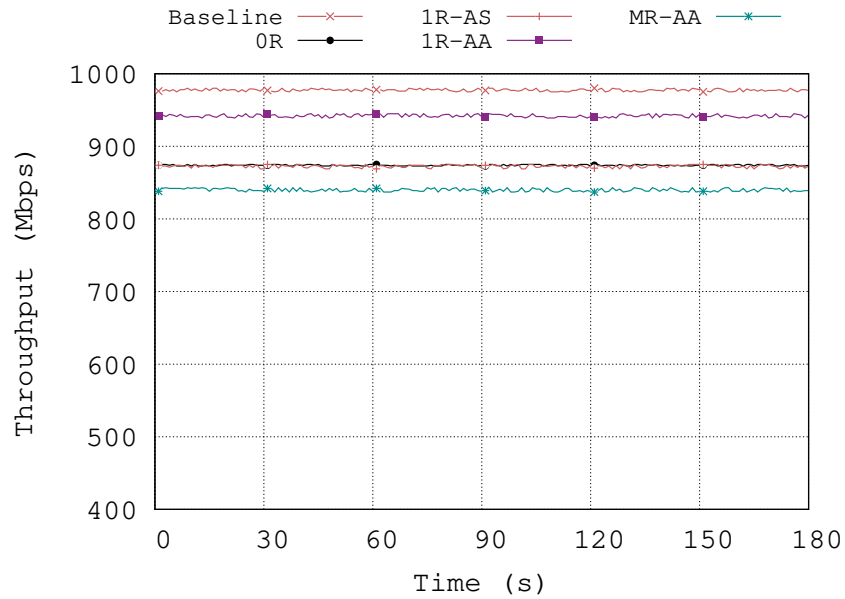


Figura 4.12: Vazão da SFC em um cenário sem falhas.

Para o mecanismo 1R-AA, a vazão diminui em aproximadamente 4,7%. O mecanismo 1R-AA realiza as suas operações diretamente em memória, melhorando significativamente a vazão da SFC. Por outro lado, o mecanismo MR-AA apresenta a maior degradação na vazão. Apesar do tempo de recuperação muito baixo, a vazão diminui em 14,1%. Assim como o 1R-AA, o mecanismo MR-AA também executa em memória, entretanto, todo o processamento feito para garantir que o grupo de réplicas de cada VNF permaneça consistente tem impacto significativo na vazão.

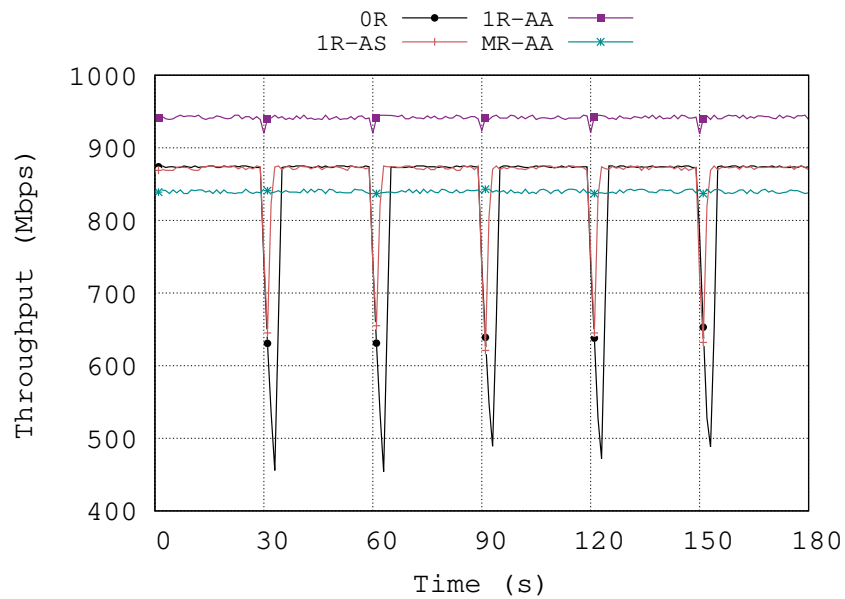


Figura 4.13: Vazão da SFC em um cenário com falhas.

No cenário com falhas na SFC, mostrado na Figura 4.13, as falhas são injetadas a cada 30 segundos. É possível observar que a redução na vazão é maior para os mecanismos 0R e

1R-AS do que para os métodos baseados em *Active-Active*, consequência dos seus maiores tempos de recuperação. Mesmo neste cenário, a vazão permanece constante para os mecanismos 1R-AA e MR-AA.

4.4.4 Avaliando a Disponibilidade das SFCs

Neste experimento, a disponibilidade de serviços baseados em NFV com o suporte do NHAM foi calculada através de um MTBF (*Mean Time Between Failures*) variável. O MTBF indica o tempo médio entre falhas. A Tabela 4.2 apresenta os resultados para cada mecanismo de resiliência. Cada experimento durou 3 horas e o MTBF indica a frequência com que as falhas foram injetadas. As SFCs nestes experimentos consistem de 8 VNFs.

Tabela 4.2: Disponibilidade da SFC conforme o MTBF varia.

MTBF	Availability (%)			
	0R	1R-AS	1R-AA	MR-AA
60 (1 min)	98.057	98.240	99.916	99.999
300 (5 min)	99.605	99.643	99.983	99.999
600 (10 min)	99.802	99.821	99.991	99.999
900 (15 min)	99.868	99.880	99.994	99.999
1200 (20 min)	99.901	99.910	99.995	99.999
1500 (25 min)	99.920	99.928	99.996	99.999
1800 (30 min)	99.934	99.940	99.997	99.999

O mecanismo 0R apresenta os níveis de disponibilidade mais baixos, embora seu custo possa justificar sua utilização para serviços que toleram tempos de recuperação mais longos. Mesmo para testes com um maior MTBF, o 0R atingiu apenas 99,3% de disponibilidade. O mecanismo 1R-AS também não atinge os níveis de disponibilidade necessários para garantir a alta disponibilidade de SFCs, apesar de alcançar resultados melhores.

Já o mecanismo 1R-AA apresenta um desempenho melhor, mesmo no cenário mais sujeito a falhas: com um MTBF de 60, as SFCs obtiveram 99,9% (*three nines*) de disponibilidade e 99,99% (*four nines*) com um MTBF de 600. Naturalmente, o mecanismo MR-AA apresenta os melhores resultados, atingindo uma disponibilidade de 99,999% (*five nines*) em todos os casos, o nível exigido de sistemas comerciais de telecomunicações.

Em geral, as plataformas em nuvem utilizadas para ambientes NFV oferecem uma disponibilidade de aproximadamente 99,9% (*three nines*) (Han et al., 2017). Os resultados desta seção permitem concluir que o NHAM consegue garantir a alta disponibilidade tanto de VNFs quanto de SFCs através dos métodos baseados no método *Active-Active*, melhorando as taxas de disponibilidade das plataformas NFV disponíveis atualmente.

4.5 TRABALHOS RELACIONADOS

Um *framework* é proposto em (Kulkarni et al., 2018) para oferecer suporte à resiliência para VNFs e SFCs. O *framework*, denominado REINFORCE, replica constantemente o estado de uma VNF para réplicas executadas localmente ou em outras infraestruturas. Para reduzir a sobrecarga causada pelo gerenciamento de estado, a replicação no REINFORCE é feita de maneira assíncrona, *i.e.*, as VNFs mantêm a sua execução enquanto o estado é sincronizado com as réplicas. Enquanto o NHAM utiliza diferentes mecanismos de resiliência, o REINFORCE utiliza apenas uma única estratégia *Active-Standby*, dificultando a adoção da solução para VNFs

com diferentes requisitos de disponibilidade. Além disso, o REINFORCE exige que os próprios desenvolvedores das VNFs indiquem quais operações causam mudanças em seu estado interno, uma tarefa não trivial e que levar a erros. Por fim, apesar de o REINFORCE ser proposto no contexto de NFV, a solução não é compatível com a arquitetura NFV-MANO, dificultando a sua integração com as plataformas NFV existentes que seguem as especificações da ETSI.

FTC (*Fault Tolerant Chain*) (Ghaznavi et al., 2020) é uma solução para melhorar a resiliência de SFCs que não utiliza *checkpoints* ou retransmissão de pacotes. Ao invés, o FTC envia as informações do estado interno da VNF no próprio pacote que é propagado através da SFC (*i.e.*, piggybacking). Cada VNF atua como uma réplica da VNF predecessora, evitando a necessidade de usar réplicas dedicadas. Após a falha de uma VNF, ela deve ser instanciada novamente e o seu estado deve ser obtido através da VNF sucessora no SFP. Diferente do NHAM, o FTC não é compatível com a arquitetura SFC proposta pela SFC, uma vez que os autores consideram que cada VNF envia o tráfego diretamente para a próxima VNF na SFC. Assim como no REINFORCE, o desenvolvedor deve modificar o código-fonte da VNF para indicar quais operações causam mudança no estado da VNF. Similar ao NHAM, o FTC também mantém o maior número de sequência recebido de cada VNF, além de um *buffer* para armazenar pacotes até que todo o estado tenha sido replicado até a última VNF da SFC. A diferença é que o *buffer* é mantido no final da SFC.

Em (Cully et al., 2008) os autores propõem Remus, um sistema para prover alta disponibilidade para máquinas virtuais que executam em hardware genérico. O Remus salva periodicamente *checkpoints* de uma máquina virtual para uma réplica. Dessa forma, se o sistema detecta uma falha, a réplica pode assumir a execução. Além disso, o Remus sincroniza os *checkpoints* mantendo os pacotes em *buffers* até que a sincronização de um novo estado seja concluída. A combinação do gerenciamento de *buffers* e *checkpoints* é semelhante ao NHAM, embora os contextos em que são empregados sejam diferentes. Apesar da diminuição no número de pacotes perdidos em caso de falha, a latência é prejudicada em situações em que não há falhas.

No trabalho proposto em (Li et al., 2015), os autores utilizam *containers* Linux para garantir a alta disponibilidade de aplicações em nuvem. A estratégia baseia-se em um *middleware* que realizam periodicamente o *checkpoint* do estado interno das aplicações. Em caso de falha, o *middleware* tenta reinicializar o *container*, o que pode comprometer o tempo de recuperação caso essa abordagem não tenha sucesso. Essa estratégia pode apresentar um alto tempo de recuperação, uma vez que se a reinicialização falhar, outra instância deverá ser criada antes importar o estado salvo. Além disso, os autores não propõem estratégias para diminuir a sobrecarga da sincronização dos estados, ou diferentes níveis de resiliência para diferentes tipos de VNFs.

Em outra proposta baseada em *buffers* (Rajagopalan et al., 2013a), os autores propõem o *Pico Replication* (PR), um *framework* de alta disponibilidade para *middleboxes*. Ao invés de capturar o estado interno dos *middleboxes*, o PR executa *checkpoints* diretamente em fluxos de dados específicos, enquanto o *middlebox* continua a processar outros fluxos. O PR requer extensas modificações para garantir a alta disponibilidade de *middleboxes*, incluindo mudanças de *kernel* e no controlador SDN.

Em (Rajagopalan et al., 2013b) os autores propõem Split/Merge, uma abstração no nível de *hypervisor* para *middleboxes* virtuais. Nessa abstração o estado da função pode ser dividido entre interno e externo através de APIs que permitem atualizar e replicar o estado entre as funções virtuais. Neste trabalho os autores implementam o sistema FreeFlow, uma implementação da abstração Split/Merge. O sistema FreeFlow separa fluxos específicos de tráfego (*e.g.*, baseados em endereço IP de origem/destino) entre réplicas, balanceando a carga entre elas. Entretanto, a implantação deste sistema demanda extensas modificações na rede, uma vez que o desenvolvedor de cada função precisa identificar, separar e gerenciar o seu próprio estado interno e os fluxos.

Em (Kablan et al., 2017) e (Khalid e Akella, 2019) são propostas soluções que desacoplam o estado interno das VNFs do seu processamento, armazenando o estado interno em um banco de dados distribuído. Dessa forma, a tolerância a falhas é alcançada para funções de rede *stateful*, uma vez que em caso de falha a nova instância pode recuperar o estado atualizado do banco de dados. Apesar dos autores de ambos os trabalhos afirmarem que as suas soluções introduzem apenas uma pequena latência para cada pacote processado, a vazão pode ser significativamente reduzida em situações onde não há falha. Além disso, a proposta não introduz nenhum tipo de redundância para as VNFs, exigindo a criação de uma nova instância em casos de falha, afetando o tempo total de recuperação. Ainda, as extensas mudanças no código interno das funções limitam esta solução, além de dificultar a integração com outras plataformas.

Outro *framework* proposto em (Harchol et al., 2018) descreve uma estratégia para garantir a tolerância a falhas de VNFs com o objetivo de minimizar a sobrecarga durante a execução sem falhas. O *framework*, denominado de FTvNF, cria duas instâncias para cada VNF de uma SFC em um modelo *master* e *slave*. Em casos de falha, o tráfego é reconfigurado para ser encaminhado para a VNF *slave*, enquanto a VNF *master* está em processo de recuperação. Para garantir a consistência entre o *master* e o *slave*, os pacotes processados pela VNF *master* são armazenados em um *logger*, enquanto que, de maneira assíncrona, a VNF *slave* recupera os pacotes armazenados do *logger*. No entanto, a implementação do *framework* FTvNF foi baseada em Click (Kohler et al., 2000), o que pode dificultar a sua implantação em ambientes reais, além de não suportar a execução de outras VNFs baseadas em outros sistemas operacionais (e.g., Ubuntu). Ainda, o FTvNF não fornece um conjunto completo de funcionalidades para garantir a alta disponibilidade de VNFs.

Uma abordagem baseada em *rollback-recovery* é proposta em (Sherry et al., 2015). O FTMB (*Fault-Tolerant MiddleBox*) é um sistema que salva o estado de *middleboxes* através de dois mecanismos: *ordered logging* e *parallel release*. O mecanismo *ordered logging* é utilizado para salvar informações necessárias para reproduzir as entradas no sistema em caso de falha. Já o mecanismo *parallel release* é um algoritmo que complementa o *ordered logging*, de modo a garantir a reprodução correta das entradas no sistema, visto que os pacotes possuem dependências entre si. Apesar dessa solução reduzir a sobrecarga em casos de falha, a estratégia proposta requer extensas modificações no código-fonte da VNF, representando uma grande limitação.

PLOVER (Wang et al., 2018) é um sistema baseado em SMR aplicado para tolerância a falhas em VMs. Neste trabalho os autores apresentam o VSMR (*Virtual SMR*), uma abordagem que garante a ordem em que os pacotes são processados em uma VM e suas respectivas réplicas. Além disso, o procedimento *syncvm* é utilizado para atualizar somente as páginas de memória necessárias. Para tanto, o PLOVER compara as páginas de memória da VM com as das réplicas, transferindo apenas as páginas divergentes.

Em (Gember-Jacobson et al., 2014) uma arquitetura de um plano de controle é proposta para redirecionar o tráfego de pacotes de uma VNF falha para outras VNFs, além de garantir a sincronização do estado interno das VNFs. O plano de controle proposto, denominado de OpenNF, gerencia o estado e minimiza a perda de informação através da transferência dos fluxos de pacotes pelo controlador. Além disso, o OpenNF propõe uma API para o gerenciamento do estado de VNFs. Entretanto, assim como as demais soluções, o OpenNF exige mudanças no código interno da VNF. Por fim, a estratégia introduz alta latência para cada pacote processado, prejudicando o desempenho global dos serviços.

No trabalho proposto em (Woo et al., 2018) os autores introduzem uma estratégia para *autoscaling* em que os estados das funções de rede são organizados através de uma abstração chamada de DSO (*Distributed Shared Object*). Nesta estratégia, objetos encapsulam o estado da função que é armazenado em um espaço de memória em que todas as instâncias das VNFs podem

ler e escrever para qualquer objeto, através de uma API unificada. Neste contexto é proposto o S6, um *framework* que estende os conceitos de DSO para o desenvolvimento e execução de VNFs.

A Tabela 4.3 compara as soluções descritas nesta seção de acordo com as seguintes propriedades: (i) virtualização, especificando os tipos de virtualização suportados pela solução; (ii) NFV, se a solução é compatível com a arquitetura de referência NFV-MANO; (iii) SFC, se a solução também garante a alta disponibilidade de SFCs; (iv) SFC IETF, se a solução além de suportar SFCs também é compatível com a arquitetura de SFCs especificada pela IETF; (v) necessidade de modificações no código-fonte; (vi) qual o método de redundância oferecido pela solução e; (vii) qual a estratégia foi utilizada pela solução.

Tabela 4.3: Comparação das soluções que fornecem alta disponibilidade para serviços e funções virtualizadas.

Solução	Virtualização	NFV	SFC	SFC IETF	Modificações no Código	Método de Redundância	Estratégia
NHAM	VM; Container	✓	✓	✓	✗	AA; AS	C/R; Buffering
REINFORCE (Kulkarni et al., 2018)	Container	✗	✓	✗	✓	AS	C/R; Buffering
FTC (Ghaznavi et al., 2020)	Click	✗	✓	✗	✓	AA	Piggybacking; Buffering
Remus (Cully et al., 2008)	VM	✗	✗	✗	✗	AS	C/R; Buffering
HA container (Li et al., 2015)	Container	✗	✗	✗	✗	Nenhum	C/R
PR (Rajagopalan et al., 2013a)	VM	✗	✗	✗	✓	AS	C/R no fluxo de dados
FreeFlow (Rajagopalan et al., 2013b)	VM	✗	✗	✗	✓	AS	Desacoplamento do estado interno
CHC (Khalid e Akella, 2019)	Container	✓	✓	✗	✓	Nenhum	Desacoplamento do estado interno
StatelessNF (Kablan et al., 2017)	Container	✗	✗	✗	✓	Nenhum	Desacoplamento do estado interno
FTvNF (Harchol et al., 2018)	VM	✓	✓	✗	✓	AA	Logger; Packet replay
FTMB (Sherry et al., 2015)	VM; Container	✓	✗	✗	✓	AS	Logger; Packet replay
PLOVER (Wang et al., 2018)	VM	✗	✗	✗	✗	AA	SMR
OpenNF (Gember-Jacobson et al., 2014)	VM; Container	✓	✗	✗	✓	AS	Desacoplamento do estado interno
S6 (Woo et al., 2018)	VM; Container	✓	✗	✗	✗	Autoscaling	Distributed Shared Object

Na maior parte das soluções descritas nesta seção, o desenvolvedor/fornecedor da VNF ou SFC necessita, ou, modificar o seu código-fonte, ou indicar no código-fonte quais operações causam alterações no estado interno da função. Esses requisitos não somente limitam a generalidade da solução, como também representam uma abordagem propensa a erros, uma vez que são feitas de maneira manual. Em (Khalid et al., 2016) os autores propõem um *framework* denominado StateAlyzr. O *framework* utiliza algoritmos de análise de programas para identificar automaticamente todo o tipo de estado que precisa ser capturado e copiado para outras réplicas. Essa abordagem pode simplificar esta etapa, reduzindo a chance de erros durante neste processo.

Além dos trabalhos descritos acima, a maioria das plataformas NFV e de nuvem, tais como OpenStack (OpenStack, 2019), OSM (ETSI, 2019), e Kubernetes (Bernstein, 2014), fornecerem mecanismos de tolerância a falhas insuficientes – não possuem estratégias para preservar o estado interno dos dispositivos virtuais, portanto não garantem a alta disponibilidade de VNFs/SFCs *stateful*.

Em geral, os trabalhos existentes fornecem somente suporte parcial para garantir a alta disponibilidade de VNFs/SFCs *stateful*. Em particular, é possível identificar três desvantagens das soluções atuais. A primeira é a falta de diferentes mecanismos de resiliência para adaptar diferentes tipos de serviços. A segunda desvantagem é que grande parte das soluções exigem extensas modificações no código-fonte da VNF, limitando tanto a solução quanto os tipos de serviços que podem ser executados na plataforma. Por fim, nenhuma das soluções é totalmente compatível com a arquitetura de referência NFV-MANO proposta pela ETSI nem com a arquitetura padrão de SFCs proposta pela IETF, dificultando a sua integração com outros sistemas NFV.

4.6 CONCLUSÕES PARCIAIS

Este capítulo propõe o NHAM: uma arquitetura de alta disponibilidade para NFV, definida como um módulo da arquitetura de referência NFV-MANO. O NHAM garante a alta

disponibilidade tanto de VNFs quanto de SFCs *stateful*, sem a necessidade de modificações no código-fonte das funções de rede. Quatro mecanismos de resiliência diferentes são definidos, os quais podem ser escolhidos de acordo com os recursos e requisitos dos diferentes tipos de VNFs. O NHAM utiliza um gerenciador de estados que combina técnicas de *Checkpoint/Restore* com o gerenciamento de *buffers* de uma SFC, permitindo a recuperação de serviços virtualizados *stateful*, mesmo após a falha simultânea de múltiplas VNFs. Desta forma, o NHAM garante a recuperação completa e correta do serviço ponto-a-ponto.

Um protótipo foi implementado e resultados experimentais foram executados para avaliar o desempenho e a disponibilidade de VNFs e SFCs com o suporte do NHAM. Os resultados mostram que o NHAM é uma solução eficaz para melhorar a robustez dos serviços virtualizados que podem atingir níveis de disponibilidade similares aos de sistemas comerciais de telecomunicações.

5 UMA ARQUITETURA PARA SFCS TOLERANTES A FALHAS E INTRUSÃO

No Capítulo 4 foram propostas estratégias para garantir a alta disponibilidade de serviços virtualizados. A estratégia assume que as VNFs que constituem um serviço virtualizado falham por parada. Por outro lado, tais serviços, em geral, são executados sobre uma arquitetura subjacente, normalmente compostas por um conjunto de componentes internos, tais como o SC e o SFF, descritos no Capítulo 2. Consequentemente, a falha em qualquer componente interno da SFC também pode comprometer toda a infraestrutura, negando o acesso a diversos serviços ou até mesmo possibilitando o uso não autorizado. Neste contexto, este capítulo propõe a FIT-SFC (*Fault- & Intrusion- Tolerant SFC*): uma arquitetura para suportar serviços virtuais seguros e altamente disponíveis, tolerando não somente falhas por parada, como também falhas bizantinas.

O restante deste Capítulo está organizado da seguinte forma. A Seção 5.1 apresenta uma breve descrição do contexto em que a FIT-SFC é proposta, além de discutir alguns trabalhos relacionados. A Seção 5.2 descreve em detalhes a arquitetura FIT-SFC, apresentando as estratégias adotadas para tolerar falhas em qualquer componente da arquitetura SFC. Em seguida, a Seção 5.3 apresenta a implementação de um protótipo e discute os resultados experimentais, enquanto que a Seção 5.4 conclui o capítulo.

5.1 FALHAS E INTRUSÕES EM COMPONENTES DA ARQUITETURA SFC

A arquitetura de referência para SFCs proposta pela IETF (descrita na Seção 2.3) é utilizada para permitir a execução de serviços de rede virtualizados. Tal arquitetura possui diversas vantagens, permitindo, por exemplo, que os serviços virtualizados sejam executados no núcleo da rede considerando inclusive diferentes tipos de topologias. Ainda, o processo de criação e configuração de novos serviços é flexibilizado e simplificado, além de facilitar a integração com sistemas baseados na arquitetura NFV-MANO.

Apesar das diversas vantagens, a tolerância a falhas está fora do escopo das especificações definidas pela IETF. No entanto, a falha nos componentes internos da SFC (*e.g.*, SC, SFF) podem comprometer o funcionamento global do serviço. Por exemplo, a falha em um componente de encaminhamento pode impedir que um fluxo de pacotes seja encaminhado entre as VNFs da SFC. O problema se agrava ao considerar que alguns componentes, tais como o SFF, podem ser distribuídos e compartilhados – consequentemente impactando múltiplos serviços.

Apesar de serviços virtualizados de rede serem amplamente explorados na literatura, tais aspectos relacionados à tolerância a falhas ainda são marginalmente abordados. Dentre as estratégias para a construção de serviços virtuais tolerantes a falhas, é possível citar aquelas que se dedicam a identificar, tolerar e, eventualmente, recuperar falhas por parada em instâncias de VNF (Ghaznavi et al., 2020; Khalid e Akella, 2019; Kulkarni et al., 2018), mas não consideram falhas nos componentes da própria arquitetura SFC.

Outra solução proposta em (Wang et al., 2021) assume a utilização de instâncias em modo *standby* para prover tolerância a falhas em serviços de rede virtualizados. Essa proposta emprega dois métodos de *Deep Reinforcement Learning* (DRL), cujo objetivo é decidir, de maneira eficiente, o mapeamento e a implantação de instâncias de VNF (tanto as ativas, quanto as *standby*) no substrato físico. Além disso, para SFCs que possuem VNFs *stateful*, um mecanismo extra é empregado para enviar constantemente o estado de uma instância de VNF ativa para as suas réplicas em *standby*. Vale ressaltar que essa estratégia de replicação de estado de VNFs

stateful é bastante custosa em termos de mensagens trocadas no sistema. Por fim, esta solução também não é nativamente compatível com a arquitetura SFC IETF.

Outro trabalho propõe Necklace (Esposito et al., 2020), uma arquitetura para a instanciação de SFCs robustas. O Necklace utiliza um algoritmo de consenso distribuído para garantir um bom desempenho do serviço e tolerar falhas de processos ou nos próprios enlaces de comunicação. A arquitetura, no entanto, tolera somente falhas não bizantinas. Já em (Kong et al., 2017a), outra solução para garantir a alta disponibilidade de SFCs é proposta. Nesse caso, os autores consideram a utilização de *backups* tanto para os enlaces quanto para as instâncias relacionadas às VNFs de um serviço. Assim, através de um algoritmo heurístico, a solução define a quantidade de réplicas a serem criadas para cada VNF presente em uma topologia completa.

Ainda, em comparação com o capítulo anterior, o NHAM utiliza estratégias baseadas em *checkpoints* de VNFs junto ao gerenciamento de *buffers* para permitir a falha de múltiplas VNFs de uma SFC. Apesar de o NHAM ser totalmente compatível com a arquitetura de referência definida pela ETSI, ele não tolera falhas nos demais componentes da SFC (*i.e.*, SC e SFF), além de garantir somente a alta disponibilidade de serviços que falham por parada.

Neste contexto, este capítulo propõe a FIT-SFC: uma arquitetura tolerante a falhas para suportar serviços virtuais seguros e altamente disponíveis. A FIT-SFC utiliza estratégias de replicação para tolerar tanto falhas por parada quanto falhas por omissão ou intrusão (*i.e.*, falhas bizantinas). A arquitetura FIT-SFC foi projetada de maneira totalmente compatível com a arquitetura de referência proposta pela IETF, possibilitando a interoperabilidade da plataforma com diversos tipos de serviços virtualizados.

A arquitetura proposta tolera falhas tanto em fluxos *stateless* quanto em fluxos *stateful*. Por sua vez, há dois tipos de consistência para fluxos *stateful*. No primeiro tipo, a FIT-SFC garante a consistência de mensagens de cada fluxo que processam. No segundo tipo, as múltiplas réplicas devem se manter consistentes tendo em vista todas as mensagens de múltiplos fluxos. Neste último caso é necessário utilizar o consenso bizantino para garantir a consistência.

A próxima seção descreve em detalhes a arquitetura FIT-SFC, apresentando as estratégias de replicação para tornar os serviços altamente disponíveis e seguros.

5.2 FIT-SFC: FAULT- & INTRUSION- TOLERANT SFC

Esta seção propõe a FIT-SFC, uma arquitetura tolerante a falhas para suportar serviços virtuais seguros e altamente disponíveis. A arquitetura FIT-SFC foi projetada de forma a permitir que a arquitetura de referência SFC proposta pela IETF tolere falhas por parada, falhas por omissão ou falhas bizantinas. Uma falha bizantina é uma falha arbitrária, normalmente utilizada para representar um componente que sofreu uma intrusão decorrente de um ataque (Castro et al., 1999) – uma função ou componente malicioso pode realizar modificações não autorizadas em pacotes que trafegam pela SFC, injetar pacotes na rede, ou ainda um componente pode tentar deixar as funções replicadas em estados inconsistentes.

Para evitar tais problemas, a estratégia utilizada pela FIT-SFC é baseada na replicação dos principais componentes da arquitetura SFC da IETF: SC, SFF e VNFs. Para permitir até f falhas bizantinas (Lamport et al., 2019), são necessárias $3f + 1$ réplicas de cada componente (mais detalhes na Subseção 5.2.1). Além disso, a arquitetura proposta evita que pacotes sejam perdidos na presença de falhas, garantindo a consistência do serviço. O modelo de sistema adotado é parcialmente síncrono, com tempo de estabilização global (GST - *Global Stabilization Time*).

A FIT-SFC, cujo fluxo de execução é ilustrado na Figura 5.1, foi, como já mencionado, projetada de maneira totalmente compatível com a arquitetura de referência da IETF. Conforme

mostra a figura, a FIT-SFC também consiste de uma sequência de N VNFs, onde cada VNF precede e sucede um SFF. A SFC pode ser constituída por um único SFF centralizado, ou pode ser constituído de múltiplos SFFs compartilhados através de um ambiente distribuído e federado. O tráfego é recebido pelo SC através de uma determinada origem (etapa 1) e é encaminhado ao SFF após ser classificado (etapa 2). Dependendo da natureza do serviço virtual, o tráfego pode consistir de um único pacote contendo uma requisição, ou pode consistir de um fluxo de pacotes. Apesar da utilização de réplicas, os pacotes seguem o fluxo padrão em que o SFF envia, para cada VNF, os pacotes a serem processados. Em seguida, a VNF envia o tráfego processado novamente ao SFF (etapa 3). Por fim, o SFF envia o tráfego ao destino (etapa 4).

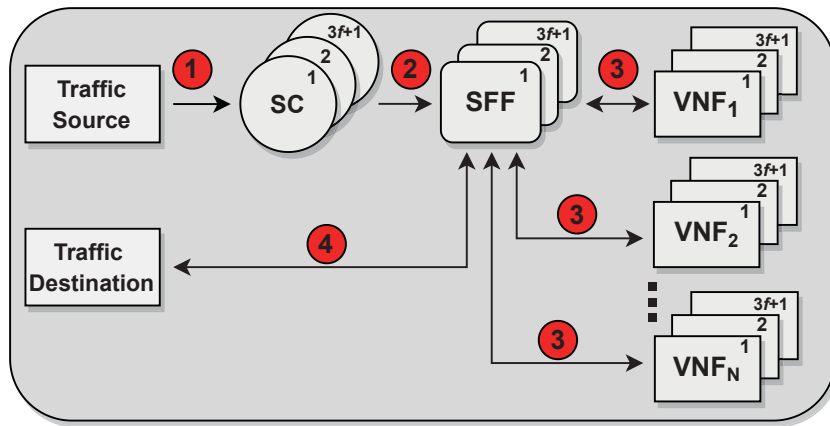


Figura 5.1: Fluxo de execução da FIT-SFC.

A subseção seguinte descreve cenários com falhas bizantinas, inclusive a necessidade de se utilizar $3f + 1$ réplicas para cada componente. Já a Subseção 5.2.2 descreve o funcionamento da arquitetura, enquanto que a Subseção 5.2.3 apresenta as diversas formas de garantir a consistência entre as múltiplas réplicas de VNFs de acordo com o tipo de fluxo processado por estas funções.

5.2.1 Cenários com Falhas Bizantinas

Considerando que n é a quantidade total de réplicas de um componente do sistema, e f é a quantidade de réplicas com falhas bizantinas, é necessário que $n \geq 3f + 1$. Considere o elemento que recebe a saída do componente replicado e deve decidir pela mensagem correta a partir das mensagens vindas das réplicas. Como o modelo de falhas bizantinas inclui a omissão, deve ser possível tomar a decisão a partir de $n - f$ mensagens, pois f mensagens podem ser omitidas. Por outro lado, é possível que nenhuma omissão tenha acontecido e que entre as $n - f$ mensagens recebidas das réplicas, f tenham sido produzidas por componentes maliciosos. Portanto, dentro das $n - f$ mensagens é necessário ter uma maioria de mensagens vindas de réplicas corretas, para garantir a decisão correta. Isso só é possível se $(n - f) - f > f$, portanto $n > 3f$ ou $n \geq 3f + 1$.

A Figura 5.2 mostra, através de um exemplo, que $n \geq 2f + 1$ réplicas são insuficientes para garantir a tolerância a falhas bizantinas – no exemplo, $n = 3$ e $f = 1$. Apenas uma das réplicas do primeiro componente é maliciosa, mas uma das réplicas corretas está lenta. O segundo componente deve tomar a decisão com somente duas mensagens (uma poderia ter sido omitida). No entanto, como existe uma réplica maliciosa, o segundo componente não chega a um quórum.

A Figura 5.3 traz outro exemplo mostrando que $n \geq 2f + 1$ réplicas são insuficientes, com $n = 3$ e $f = 1$. No caso, o cliente é malicioso e está em conluio com uma réplica do primeiro

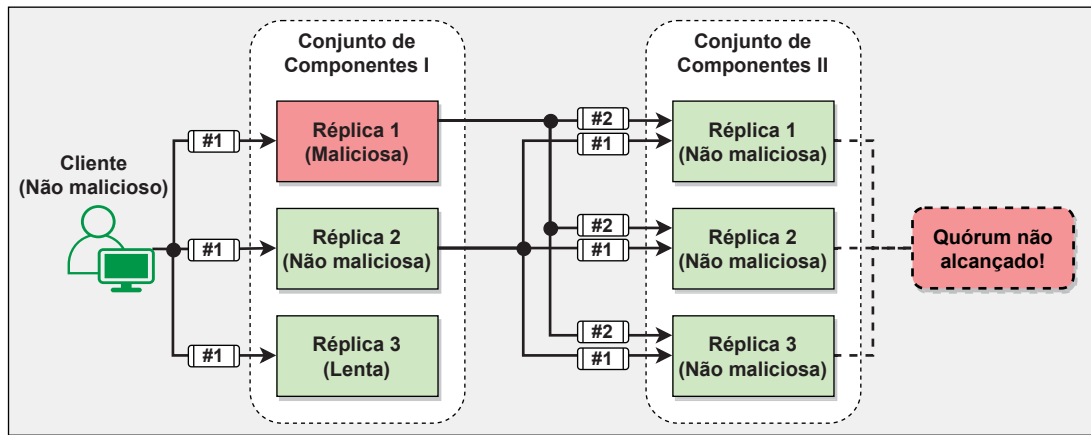


Figura 5.2: Cenário de impossibilidade de quórum (falhas bizantinas).

componente. Para as réplicas do primeiro componente, o cliente malicioso envia a mensagem #1 para uma réplica correta e a mensagem #2 para a outra réplica correta. A réplica maliciosa conhece ambas as mensagens. As réplicas do segundo componente recebem as mensagens #1 e #2 das réplicas corretas do primeiro componente, mas a réplica maliciosa encaminha a mensagem #1 para uma réplica correta, a mensagem #2 para outra réplica correta, além de omitir a mensagem que seria enviada para a última réplica correta. Nesta situação, uma réplica correta obtém quórum para a mensagem #1, enquanto a outra réplica correta obtém quórum para a mensagem #2, ainda, a Réplica 3 não obtém quórum. Dessa forma, em (Lamport et al., 2019), é provado que o número de réplicas deve ser de pelo menos $3f + 1$ para obter um quórum de pelo menos $2f + 1$ mensagens idênticas.

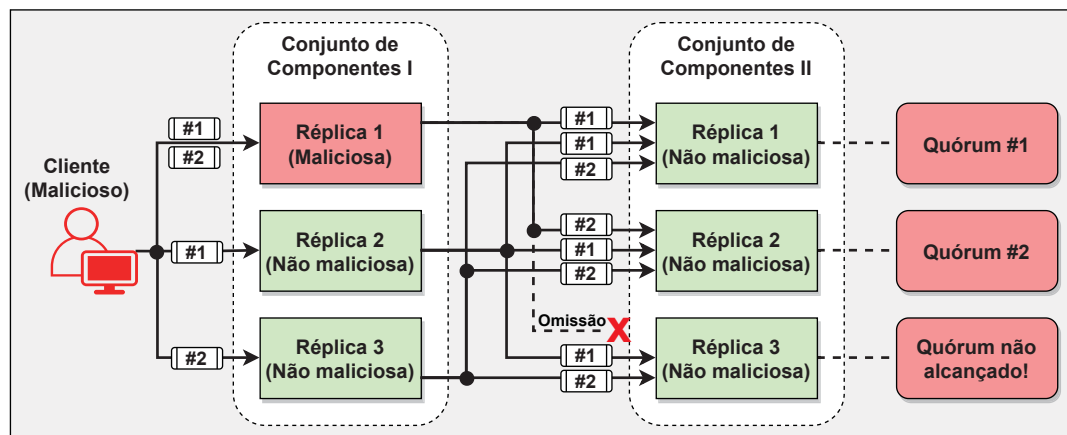


Figura 5.3: Cenário de conluio com cliente malicioso.

5.2.2 Funcionamento da Arquitetura FIT-SFC

Um cliente deve utilizar um *wrapper* para encaminhar requisições para um serviço FIT-SFC. O *wrapper* tem duas finalidades: (i) enviar uma cópia de cada pacote para cada uma das $3f + 1$ réplicas do SC e; (ii) marcar cada pacote enviado com um *timestamp* local, que pode ser implementado como um contador local de pacotes que serão transmitidos para a SFC, cujo objetivo é permitir a identificação única de cada pacote.

O tráfego é então recebido por cada uma das réplicas do SC (denominada de *FIT-Classifier*). Cada uma delas processa e encaminha uma única vez cada pacote recebido para cada

uma das $3f + 1$ réplicas de SFF. Como pode ser observado na Figura 5.4, antes do SFF realizar o encaminhamento do tráfego entre as VNFs do SFP correspondente, é necessário considerar que podem haver até f instâncias da *FIT-Classifer* falhas ou que a origem do tráfego é maliciosa. Portanto, cada SFF deve realizar uma votação para selecionar o pacote correto a ser encaminhado para a VNF replicada. É importante observar que versões diferentes de pacotes podem ser recebidas devido a falhas bizantinas.

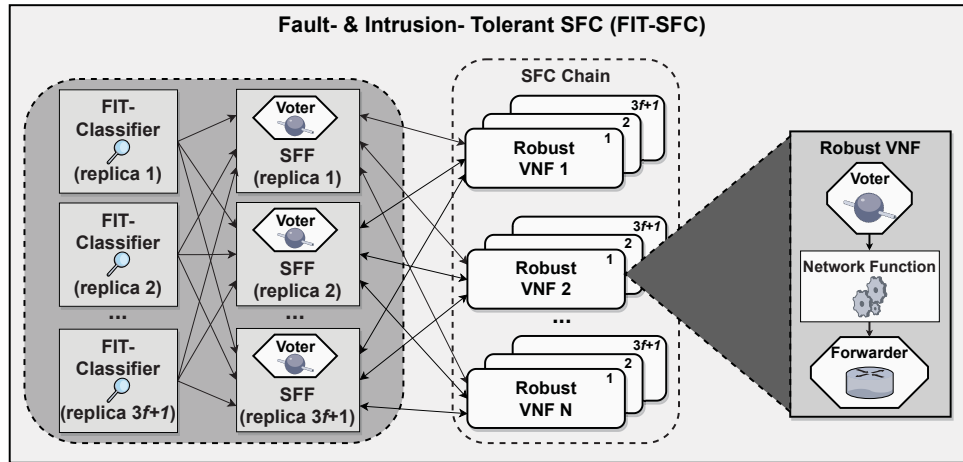


Figura 5.4: A arquitetura FIT-SFC.

Para viabilizar a votação, um elemento adicional é incorporado às réplicas de SFF, denominado de *Voter*. A partir do momento em que o *Voter* recebe $2f + 1$ cópias idênticas de um mesmo pacote (*i.e.*, um quórum), este é autorizado a processar e encaminhar o mesmo para o destino seguinte (*i.e.*, as VNFs). O número de cópias adotado pode ser entendido da seguinte forma: inicialmente, para existir uma decisão unânime entre todos os componentes de um determinado sistema, é necessário o recebimento de uma quantidade mínima de mensagens provenientes de componentes corretos, sendo que esta deve superar o número de mensagens provenientes de (ou omitidas por) componentes falhos. Ou seja, ao receber $2f + 1$ cópias do pacote, é possível que até f cópias tenham sido produzidas por componentes bizantinos. Consequentemente, $f + 1$ cópias necessariamente correspondem ao pacote original correto. Sendo assim, a estratégia de replicação garante a tolerância de f falhas de cada tipo de componente dado um número de réplicas de cada componente igual a $3f + 1$. Entretanto, se não houver um quórum de $2f + 1$ pacotes coincidentes, o encaminhamento e processamento do tráfego é imediatamente interrompido.

Em seguida, cada réplica do SFF envia a sua cópia do pacote para cada uma das réplicas da VNF, de acordo com a ordem definida no SFP. Semelhante aos SFFs, cada réplica da VNF também possui um *Voter* que aguarda por $2f + 1$ cópias idênticas de um determinado pacote, antes de efetivamente realizar o processamento da função. Da mesma forma, se menos do que esse número de cópias for recebido pelo *Voter* das VNFs, o fluxo deve ser interrompido.

Após cada réplica de VNF completar a sua execução, o subcomponente *Forwarder* envia uma cópia de cada pacote de volta para cada réplica de SFF. Nesta etapa, cada réplica de SFF realiza uma nova votação para obter um quórum de $2f + 1$ pacotes, antes de encaminhar os pacotes para as próximas réplicas da VNF de acordo com o SFP. Este processo se repete até que o tráfego tenha sido processado pela última VNF replicada da SFC. Finalmente, cada réplica do SFF envia uma cópia do pacote para o destino. Novamente, uma votação final é realizada e o pacote só é entregue no destino se houver $2f + 1$ cópias idênticas.

5.2.3 Garantindo a Consistência de VNFs Replicadas

Ao considerar VNFs replicadas, a consistência entre as múltiplas réplicas deve ser mantida. Existem diferentes tipos de funções de rede, e o tráfego que essas funções processam podem ou não modificar o seu estado interno. Se o estado interno da VNF é desacoplado do seu processamento, dizemos que esta VNF é *stateless*. Por outro lado, se o estado interno da função de rede muda de acordo com os pacotes processados, dizemos que ela é *stateful*. Para garantir a consistência do serviço, cada VNF replicada executa um protocolo diferente conforme o tipo de consistência esperado. Em particular, na arquitetura FIT-SFC os serviços de rede podem ser classificados de três formas diferentes: (i) serviços *stateless*; (ii) serviços *stateful* por fluxo e; (iii) serviços *stateful* global. Cada tipo de serviço é descrito a seguir.

Para os serviços *stateless*, uma vez que eles não possuem estado, basta realizar o processamento dos pacotes recebidos de acordo com a ordem de recebimento. Como estas funções não mantêm estado, a consistência do serviço não é comprometida se somente parte das réplicas da VNF executam um determinado pacote.

Por outro lado, funções de rede *stateful* por fluxo possuem um estado para cada fluxo percorrendo o serviço, *i.e.*, necessita de ordem parcial. Portanto, os pacotes provenientes de um mesmo fluxo devem ser processados na mesma ordem por todas as réplicas de VNFs executando uma mesma função. Este cenário é tratado na arquitetura FIT-SFC através do *timestamp* local adicionado pelo *wrapper* dos clientes, o qual é utilizado para criar um ordenamento no processamento padrão para os pacotes de um fluxo.

Por fim, as funções de rede do tipo *stateful* global recebem múltiplos fluxos de diferentes origens, e o seu estado é definido a partir do processamento de todos os pacotes recebidos. Neste caso, cada réplica precisa executar cada pacote de cada fluxo na mesma ordem, *i.e.*, necessita de ordem total. A existência de um estado único demanda que todos os pacotes sejam processados na mesma ordem por todas as réplicas de VNFs que executam uma mesma função, sendo necessário adotar um mecanismo de consenso entre as mesmas. Um exemplo clássico deste tipo de serviço é um balanceador de cargas. Esse tipo de função recebe fluxos de diferentes origens e distribui a carga para diferentes servidores. Ao replicar este tipo de serviço, é essencial garantir que todas as réplicas processam e enviam os pacotes na mesma ordem para o mesmo conjunto de servidores.

Em particular, a arquitetura FIT-SFC implementa o algoritmo de consenso PBFT (Castro e Liskov, 2002), descrito na Seção 3.4. O consenso é executado somente para os serviços *stateful* global, garantindo a ordem total entre fluxos, além de garantir que todas as réplicas da VNF executam a mesma sequência de pacotes. O consenso é implementado junto ao componente *Voter* da VNF e a execução do consenso é realizada antes da etapa de votação. Dessa forma, a entrega dos pacotes para cada *Voter* é feita na mesma ordem.

5.3 IMPLEMENTAÇÃO E RESULTADOS EXPERIMENTAIS

Esta seção apresenta uma avaliação do custo imposto pela arquitetura FIT-SFC sobre os serviços tolerantes a falhas e intrusões. Inicialmente, é apresentada a implementação de um protótipo, possibilitando a avaliação e a verificação do seu funcionamento. Para tanto, três serviços virtualizados com diferentes características também foram implementados e executados sobre a FIT-SFC: (i) um serviço de DNS (*Domain Name System*); (ii) um serviço de DNS autenticado e; (iii) um serviço de balanceamento de carga (*Load Balancer* - LB). Em seguida, são apresentados os resultados para o custo em termos da latência, comparando versões das SFCs construídas com e sem a FIT-SFC.

5.3.1 Implementação da Arquitetura FIT-SFC e dos Serviços Virtualizados

Como prova de conceito, um protótipo da arquitetura FIT-SFC foi desenvolvido. O protótipo implementa todos os elementos operacionais da arquitetura SFC da IETF (descritos na Seção 2.3) utilizando a linguagem Python 3.9. Os elementos realizam todas as trocas de mensagens descritas na Seção 5.2, tolerando até f falhas bizantinas em cada componente da arquitetura.

Além da configuração de rotas e fluxos tradicionais da arquitetura SFC da IETF, os SCs da arquitetura FIT-SFC são configurados com informações sobre as réplicas de SFFs em execução, permitindo a troca de mensagens entre elas. Por sua vez, os SFFs também são configurados para identificar, através do endereço IP, cada réplica dos SCs e das VNFs dos serviços instanciados. Essa configuração é necessária uma vez que o SFF recebe cópias de mensagens de cada réplica do SC e de cada réplica de VNF dos serviços instanciados. Por fim, as VNFs também são configuradas para identificar todas as réplicas de SFFs, permitindo tanto o envio quanto o recebimento de tráfego entre estes elementos.

Para verificar o funcionamento correto da arquitetura FIT-SFC, além de avaliar o custo para tolerar as falhas e intrusões, três serviços de rede virtualizados foram implementados: um serviço de DNS, um serviço de DNS autenticado e um serviço de balanceamento de cargas. Os serviços são brevemente descritos a seguir.

DNS. O serviço de nomes (*Domain Name System* - DNS) implementado é baseado na especificação definida no RFC 1035 (Mockapetris, 1987). O serviço recebe como argumento um domínio e retorna ao usuário as informações correspondentes. Neste caso, a SFC consiste de uma única VNF de DNS. Uma vez que o tráfego processado por essa VNF não modifica o seu estado interno, este serviço é classificado como *stateless*.

DNS autenticado. O segundo serviço implementado é um DNS autenticado. As funcionalidades são as mesmas do serviço anterior, no entanto, existe uma etapa adicional de autenticação. Neste caso, a SFC consiste de duas VNFs: uma VNF que realiza somente a autenticação, e outra que executa a funcionalidade tradicional de DNS. Este serviço é considerado *stateful por fluxo*, pois necessita armazenar informações de autenticação do usuário enquanto o fluxo está ativo.

Balanceador de cargas. O terceiro e último serviço implementado é um balanceador de cargas. O serviço implementa uma política *round-robin* em que cada requisição é distribuída de maneira sequencial e ciclicamente para um conjunto de N servidores. A SFC deste serviço consiste de uma única VNF, e o serviço é *stateful entre fluxos* (global), uma vez que cada pacote recebido altera o contador de pacotes de cada instância do balanceador de cargas. Dessa forma, para garantir a consistência entre as múltiplas réplicas da SFC, todas as instâncias do serviço devem processar os mesmos fluxos/pacotes na mesma ordem. Portanto, para este serviço, a etapa de consenso (descrita na Seção 5.2) é necessária.

5.3.2 Resultados Obtidos

Os cenários de testes da arquitetura FIT-SFC foram construídos utilizando o emulador NIEP (Tavares et al., 2018) e executados em uma máquina com processador Intel Core I7 @ 2.5GHz, 16GB RAM DDR4 e Ubuntu 16.04. Para cada experimento descrito nesta seção, dois cenários são comparados. O primeiro cenário executa os elementos operacionais da arquitetura FIT-SFC, porém utiliza apenas uma instância de cada elemento, *i.e.*, sem tolerância a falhas nem intrusões ($f = 0$). Já o segundo cenário consiste de quatro réplicas de cada elemento da arquitetura FIT-SFC, *i.e.*, com tolerância a falhas ($f = 1$). Todos os experimentos relatados

foram executados 50 vezes e as médias dos resultados são apresentadas com um intervalo de confiança de 95%.

Os experimentos descritos a seguir avaliam, para cada serviço implementado, qual o impacto em termos de latência ao utilizar a arquitetura FIT-SFC. A latência total para cada cenário (*i.e.*, com e sem replicação) considera a progressão do tempo total de processamento de um pacote do momento da emissão do mesmo pelo cliente até o seu recebimento pelo servidor ao qual foi destinado, passando antes por todos os elementos operacionais da arquitetura FIT-SFC. Além disso, cada experimento determina o tempo consumido em cada etapa do fluxo de execução: (i) SC: tempo despendido entre o cliente e os SCs; (ii) SFF: tempo despendido entre os SCs e os SFFs; (iii) NF: tempo despendido entre os SFFs e as funções de rede (*Network Function* - NF).

O primeiro experimento, ilustrado na Figura 5.5, avalia a latência para o serviço DNS. O tamanho de cada requisição é de 512 Bytes. Para o cenário sem replicação, o tempo total da requisição foi de 12,16ms (*i.e.*, Origem $\rightarrow$ SC $\rightarrow$ SFF $\rightarrow$ NF $\rightarrow$ SFF $\rightarrow$ Destino). Do tempo total, aproximadamente 65% foi dedicado a etapa SC, uma vez que o tráfego precisa ser classificado e cada pacote deve ser encapsulado com o NSH. Já as etapas de SFF e NF, ambas representam aproximadamente 17,5% do tempo total da requisição. O tempo para estas etapas é menor, visto que o SFF simplesmente encaminha pacotes (*i.e.*, não realiza processamento) e a NF atende uma requisição local.

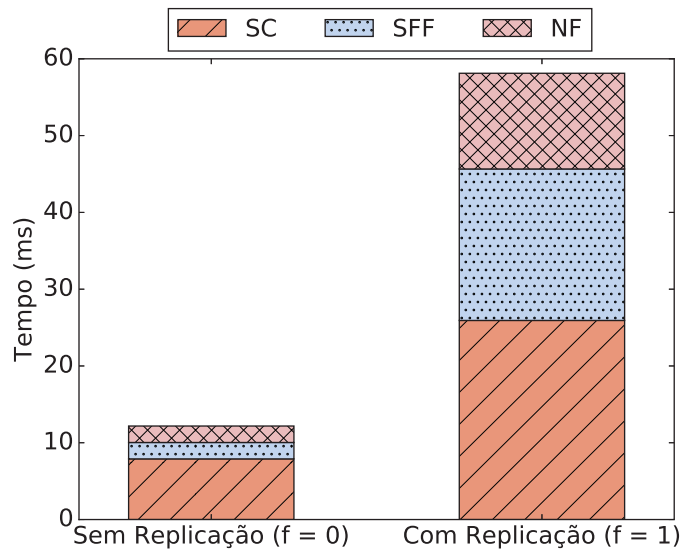


Figura 5.5: Comparação da latência do serviço de DNS.

Ao comparar com o cenário com tolerância a falhas ($f = 1$), o tempo total da requisição aumenta de 12,16ms para 58,1ms – aproximadamente 4,7 vezes maior. Esse impacto ocorre em decorrência das trocas de mensagens adicionais e das comparações que o componente *Voter* realiza para garantir que ao menos $2f + 1$ réplicas (nestes experimentos, 3 réplicas) receberam o mesmo pacote. Em particular, o tempo da etapa de SC aumenta de 7,89ms para 25,92ms, a etapa de SFF aumenta de 2,13ms para 19,72ms, enquanto que a etapa de NF aumenta de 2,143ms para 12,46ms.

O segundo experimento mede a latência para o serviço de DNS autenticado, ilustrado na Figura 5.6. O tamanho de cada requisição também é de 512 Bytes, além das informações relacionadas à autenticação. A diferença para o experimento anterior é que a SFC para o DNS autenticado é composta de duas VNFs. No cenário sem replicação, o tempo total de cada requisição é de 20,6ms – valor levemente maior em comparação com o experimento anterior devido ao passo extra de autenticação (*i.e.*, etapa NF 1) e ao encaminhamento adicional do

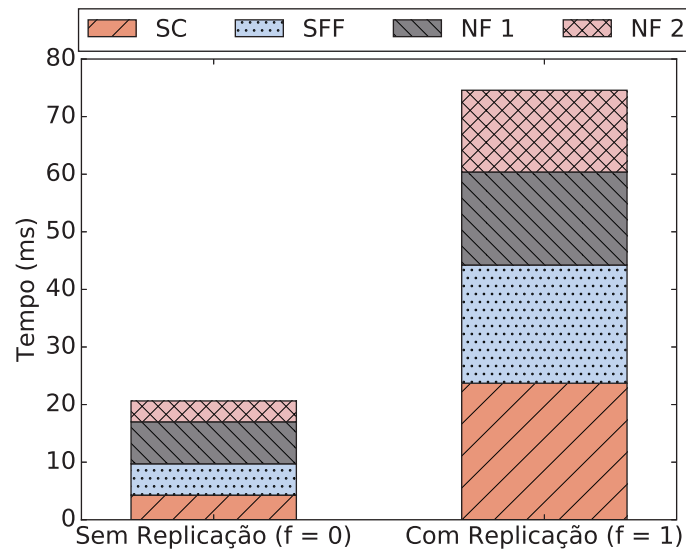


Figura 5.6: Comparação da latência do serviço de DNS autenticado.

SFF (*i.e.*, Origem $\rightarrow$ SC $\rightarrow$ SFF $\rightarrow$ NF 1 $\rightarrow$ SFF $\rightarrow$ NF 2 $\rightarrow$ SFF $\rightarrow$ Destino). Em comparação com o serviço anterior, o tempo da etapa de SFF aumenta linearmente, devido ao encaminhamento adicional com duas funções de rede.

No cenário com tolerância a falhas, o tempo total da requisição aumenta para 74,56ms – aproximadamente 3,6 vezes maior. Ao comparar com o serviço de DNS sem autenticação, houve um aumento de 16,46ms, tempo que corresponde ao processamento da função de autenticação (NF 1) e ao passo adicional entre as VNFs e o SFF.

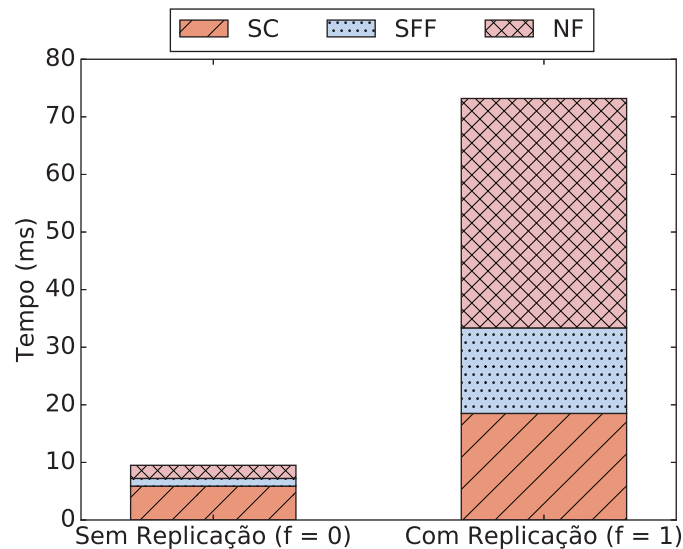


Figura 5.7: Comparação da latência do serviço de balanceamento de cargas.

Finalmente, o terceiro experimento avalia a latência do serviço de balanceamento de cargas, ilustrado na Figura 5.7. Para este experimento, o tamanho de cada requisição é de 1024 Bytes. No cenário sem replicação, o tempo total de cada requisição foi de 9,47ms, dos quais 61,9% foi destinado à etapa de classificação (SC), enquanto que 14,04% e 23,9% foram destinados às etapas de SFF e NF, respectivamente.

Comparando novamente com o cenário com replicação, o tempo total de cada requisição aumenta para 73,17ms – aproximadamente 7,7 vezes maior. A maior parte da sobrecarga imposta para esta função está no consenso, que deve ser executado para garantir a consistência entre as múltiplas réplicas do balanceador de cargas. Em particular, a etapa de NF para este cenário consome 54,4% do tempo, o que corresponde ao tempo de processamento da função, além do tempo de execução do algoritmo de consenso.

Naturalmente, o uso da arquitetura FIT-SFC tem um custo associado em termos de aumento da latência. Os componentes são replicados, aumentando a quantidade de pacotes redundantes transitando na rede e que precisam ser recebidos e processados.

5.4 CONCLUSÕES PARCIAIS

Este capítulo apresenta a FIT-SFC, uma arquitetura baseada em replicação para prover tolerância a falhas e intrusões para serviços virtualizados de rede. A FIT-SFC permite que os próprios componentes internos da SFC sofram falhas por parada, omissão ou falhas por intrusão. Para tolerar f falhas, cada componente é configurado com $3f + 1$ réplicas. Além disso, o componente *Voter* é incorporado nos SFFs e nas VNFs, responsável por comparar diferentes cópias de mensagens e, somente após receber um quórum de $2f + 1$ cópias idênticas, autorizar o processamento do pacote.

Um protótipo foi desenvolvido e avaliado através de três serviços de rede: um DNS, um DNS autenticado e um balanceador de cargas. Resultados experimentais demonstram que, apesar do custo imposto pela solução em termos de latência, serviços podem se tornar seguros e altamente disponíveis sem a necessidade de realizar extensas modificações no seu código-fonte.

6 NFV-COIN: NFV *COMPUTING IN THE NETWORK*

Nos últimos anos o conceito COIN (*COmputing In the Network*) tem se tornado importante, no que diz respeito ao processamento de serviços dentro da própria rede (Zeng et al., 2021). Por outro lado, a tecnologia NFV tem sido utilizada majoritariamente para oferecer serviços de rede tradicionais de maneira virtualizada, como *firewalls* e detectores de intrusão. Com os recentes avanços nas tecnologias de virtualização, surge uma nova possibilidade para a oferta de serviços de rede inovadores, executados na própria rede. Para tanto, este capítulo explora a sinergia entre o paradigma NFV e o conceito COIN, denominado neste trabalho de NFV-COIN.

Este capítulo está organizado da seguinte forma. A Seção 6.1 apresenta uma visão geral sobre COIN e descreve a sua relação e diferenças com NFV. A Seção 6.2 introduz o conceito NFV-COIN e descreve três casos de estudo. A Seção 6.3 por sua vez propõe uma arquitetura NFV-COIN, totalmente compatível com a arquitetura de referência NFV-MANO. Em seguida, a Seção 6.5 descreve detalhes de implementação de um protótipo da arquitetura proposta, além dos resultados experimentais obtidos. Por fim, a Seção 6.6 apresenta as conclusões parciais.

6.1 *COMPUTING IN THE NETWORK*: UMA VISÃO GERAL

Apesar de que a maioria dos serviços virtualizados implementam funcionalidades clássicas de rede, o paradigma NFV pode ser utilizado para prover serviços originais e inovativos oferecidos no núcleo da rede. Dessa forma, serviços de rede que são normalmente executados no próprio servidor do usuário, agora podem ser executados diretamente dentro da rede, no conceito que tem sido chamado COIN (Zeng et al., 2021).

O conceito COIN pode ser visto no contexto *edge-to-cloud continuum*, um conceito que explora a possibilidade do processamento distribuído em *cloud* obter níveis de desempenho em termos de baixa latência e alta vazão, segurança e alta disponibilidade próximos aos da borda da rede (*i.e.*, *edge*). COIN consegue integrar naturalmente o processamento dentro da rede (*i.e.*, *In-Network Computing* - INC) e o processamento da própria rede em um único *framework*. A própria arquitetura de redes móveis 5G e 6G devem se beneficiar do conceito COIN como uma tecnologia facilitadora para oferecer serviços diversos, inclusive aqueles no contexto de *Internet of Things* (IoT), veículos autônomos e realidade virtual distribuída.

A sinergia da união entre NFV e COIN já foi explorada anteriormente. Em 2015 (Turchetti et al., 2015), um serviço NFV-COIN de detecção de falhas clássico de sistemas distribuídos foi implementado; o trabalho foi expandido em 2017 (Turchetti et al., 2017). O consenso foi outro serviço distribuído implementado na ideia NFV-COIN (Venâncio et al., 2021), utilizado para manter um grupo de controladores SDN consistente. Ainda em outro trabalho, a rede foi habilitada para oferecer nativamente serviços de difusão confiável e ordenada de mensagens (Venâncio et al., 2019). No entanto, tais serviços não foram formalmente definidos e especificados no contexto NFV-COIN.

Uma forma alternativa de implementar serviços na própria rede é através da utilização de hardware programável (Tokusashi et al., 2019), chamados de serviços INC, geralmente focados em alto desempenho. Em 2016, Soulé *et. al.* (Dang et al., 2016, 2020) implementaram o algoritmo de consenso Paxos em um *switch* utilizando a linguagem P4 (Bosshart et al., 2014). Outros serviços INC incluem sistemas para armazenamento de dados (Bressana et al., 2020); IncBricks (Liu et al., 2017), um sistema INC para cache; e um sistema distribuído de partição/agregação

de dados (Sapio et al., 2017). Tais serviços foram implementados utilizando tecnologias como ASICs (*Application Specific Integrated Circuits*) e FPGA (*Field Programmable Gate Array*). Outros trabalhos (Misa et al., 2021; Marques et al., 2019) são propostos no contexto de serviços de telemetria de redes. Tais serviços foram implementados em *switches* programáveis para melhorar a escalabilidade da telemetria de toda a rede com relação ao tráfego dinâmico e de acordo com a carga de consulta. A Tabela 6.1 apresenta um resumo de serviços INC desenvolvidos com hardware programável ou com NFV.

Tabela 6.1: Serviços INC e NFV-COIN.

Serviço	NFV	INC	Desempenho
Detector de falhas (Turchetti et al., 2015)	✓		Redução no uso de CPU e memória
Detector de falhas (Turchetti et al., 2017)	✓		Redução no uso de CPU e memória
Consenso (Dang et al., 2016)		✓	Aumento na vazão; Redução na latência
Consenso (Dang et al., 2020)		✓	Aumento na vazão
Consenso (Venâncio et al., 2021)	✓		Aumento na vazão; Redução no tempo de resposta e uso de CPU
Cache (Liu et al., 2017)		✓	Aumento na vazão; Redução na latência
Agregação (Sapio et al., 2017)		✓	Redução no uso de banda e no tempo de processamento
Agregação (Lao et al., 2021)		✓	Aumento na vazão; Uso eficiente de recursos
Difusão confiável (Venâncio et al., 2019)	✓		<i>Trade-off</i> justo entre vazão e latência
Armazenamento (Bressana et al., 2020)		✓	Aumento na vazão; Baixa sobrecarga na latência
Triagem de transações (Jepsen et al., 2021)		✓	Aumento na vazão; Redução no tempo de processamento
Telemetria (Misa et al., 2021)		✓	Aumento na escalabilidade

Apesar de que os serviços implementados em hardware geralmente apresentam melhor desempenho, a abordagem NFV-COIN é certamente uma melhor escolha para executar sistemas mais complexos que são geralmente executados nos servidores dos usuários finais. Os serviços NFV-COIN herdam todas as vantagens da tecnologia NFV, sendo a principal delas a maior flexibilidade para implementar e gerenciar novos serviços. No entanto, existem aplicações para ambas as abordagens.

É importante ressaltar a diferença entre SFCs tradicionais e serviços COIN: as SFCs implementam serviços de rede clássicos, que sempre foram executados na rede. Por outro lado, serviços COIN são tradicionalmente executados pelo próprio usuário final, ou como *middlewares* ou como aplicações. Portanto, do ponto de vista do usuário final, a maior vantagem dos serviços NFV-COIN está na facilidade de simplesmente utilizar um serviço que já é oferecido nativamente pela própria rede, ao invés de implementar e manter localmente os serviços. Alguns serviços podem inclusive ser reprojatados para utilizar informações provenientes da rede, de forma a oferecer novas funcionalidades. Utilizando como exemplo o detector de falhas mencionado acima, pode-se utilizar informações obtidas da própria rede para detectar falhas de enlace.

No entanto, a união do paradigma NFV com o conceito COIN certamente traz desafios que devem ser explorados. Uma vez que a arquitetura de referência NFV-MANO tem sido amplamente utilizada como padrão para a execução e integração de serviços virtuais de rede, a arquitetura proposta para suportar serviços NFV-COIN deve ser totalmente compatível com aquela arquitetura. Além disso, deve também fornecer interfaces padronizadas que permitam aos usuários finais manipular os serviços de maneira sofisticada e customizada. Dessa forma, a interação entre o usuário final e os serviços NFV-COIN deve ser precisamente especificada, definindo, por exemplo, como os serviços serão oferecidos, configurados, mantidos e acessados.

Portanto, uma arquitetura para serviços NFV-COIN deve incluir (i) uma interface dentro da rede para acessar os serviços NFV-COIN e; (ii) um módulo que suporte de maneira padronizada a criação sob demanda, manipulação e oferta dos serviços NFV-COIN, além de fornecer acesso seguro e autenticado para estes serviços. Por fim, tal arquitetura deve ser especificada de maneira totalmente compatível com a arquitetura NFV-MANO, de forma a permitir a integração simplificada com múltiplas soluções NFV.

É importante, no entanto, observar a diferença entre as funcionalidades oferecidas pela arquitetura NFV-MANO e a arquitetura NFV-COIN. Enquanto a arquitetura NFV-MANO oferece funcionalidades que permitem o controle e gerenciamento típico de VNFs e SFCs, a arquitetura NFV-COIN atua em um nível acima, abstraindo as operações do NFV-MANO que são específicas do sistema subjacente e permitindo a manipulação padronizada dos serviços virtualizados. Consequentemente, mesmo que o sistema subjacente mude, as primitivas para acessar os serviços permanecerão as mesmas.

6.2 SERVIÇOS NFV-COIN: TRÊS ESTUDOS DE CASO

A Figura 6.1 representa uma arquitetura genérica de uma rede virtualizada. No topo da figura é possível observar um conjunto de servidores físicos localizados na rede. Estes servidores contêm uma camada de virtualização, responsável por abstrair os recursos físicos em recursos virtuais através de *containers* ou VMs. Neste ambiente, VNFs arbitrárias podem ser executadas. A figura mostra tanto exemplos de serviços NFV-COIN quanto de funções tradicionais de rede. Os serviços virtualizados são acessados por sua vez pelos usuários finais a partir das suas redes, representados na parte inferior da figura.

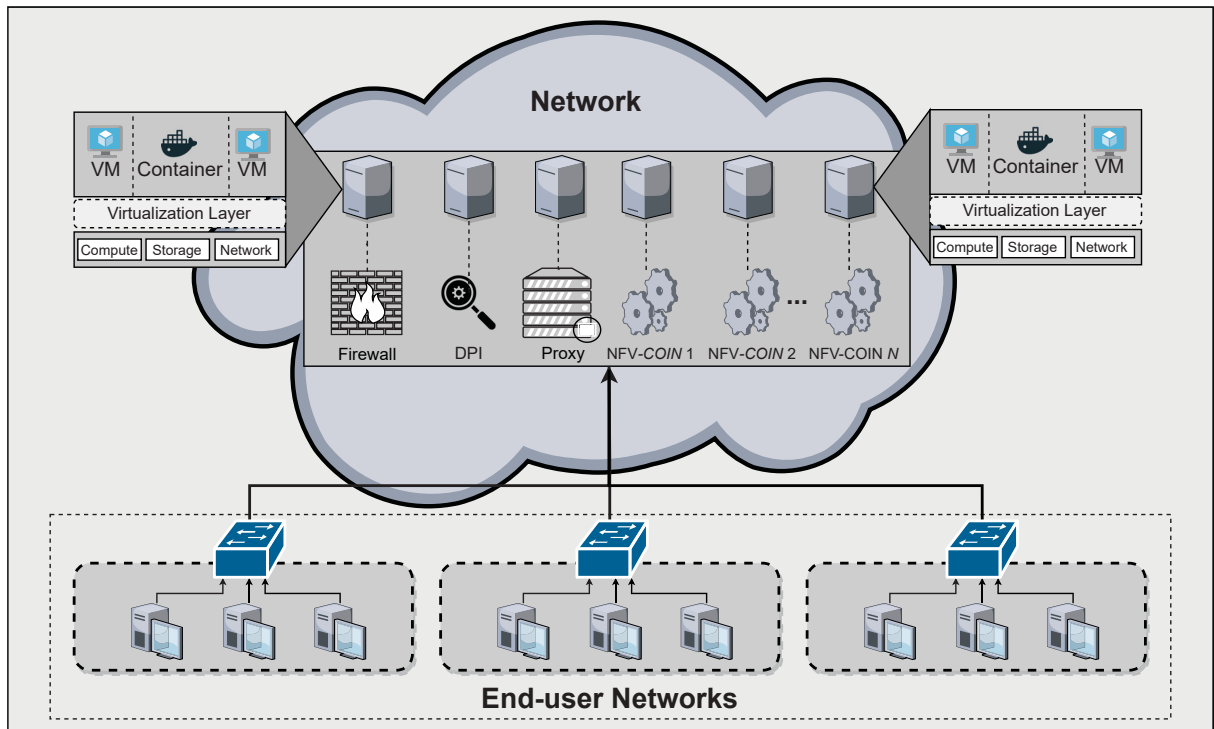


Figura 6.1: Uma rede virtualizada genérica.

O potencial da utilização de tecnologia NFV para construir serviços COIN é descrito a seguir através de três casos de estudo. O primeiro serviço NFV-COIN, descrito na Seção 6.2.1, implementa um detector de falhas. O segundo serviço, descrito na Seção 6.2.2 apresenta

a implementação do consenso como um serviço virtualizado. Por fim, o terceiro estudo de caso, descrito na Seção 6.2.3, apresenta uma estratégia para permitir a própria rede oferecer nativamente serviços de difusão confiável e ordenada de mensagens.

É importante notar que no momento em que estes serviços foram propostos, o conceito NFV-COIN como descrito neste capítulo ainda não havia sido estabelecido. No entanto, eles representam exemplos de tipos de serviços que podem se beneficiar do conceito NFV-COIN proposto neste trabalho.

6.2.1 Detector de Falha como um Serviço NFV-COIN: NFV-FD

Como visto na Seção 3.1, detectores de falhas foram originalmente propostos para auxiliar na solução do consenso em sistemas distribuídos assíncronos em que os processos falham por parada. Um detector de falhas baseado em NFV foi proposto em (Turchetti et al., 2017), executado como um serviço NFV-COIN. A proposta denominada de NFV-FD (NFV *Failure Detector*) é ilustrada na Figura 6.2. O NFV-FD utiliza informações obtidas de um controlador SDN – neste caso, um controlador OpenFlow (McKeown et al., 2008) – para monitorar os processos do sistema e determinar o seu estado (*i.e.*, correto ou suspeito de falha).

Um dos principais componentes do NFV-FD é um filtro de pacotes chamado FMod, aplicado somente nos pacotes recebidos pelo controlador e responsável por selecionar somente os pacotes que possuem informações sobre os processos monitorados. O FMod inspeciona os pacotes examinando campos do cabeçalho e envia parte da informação para os processos monitorados.

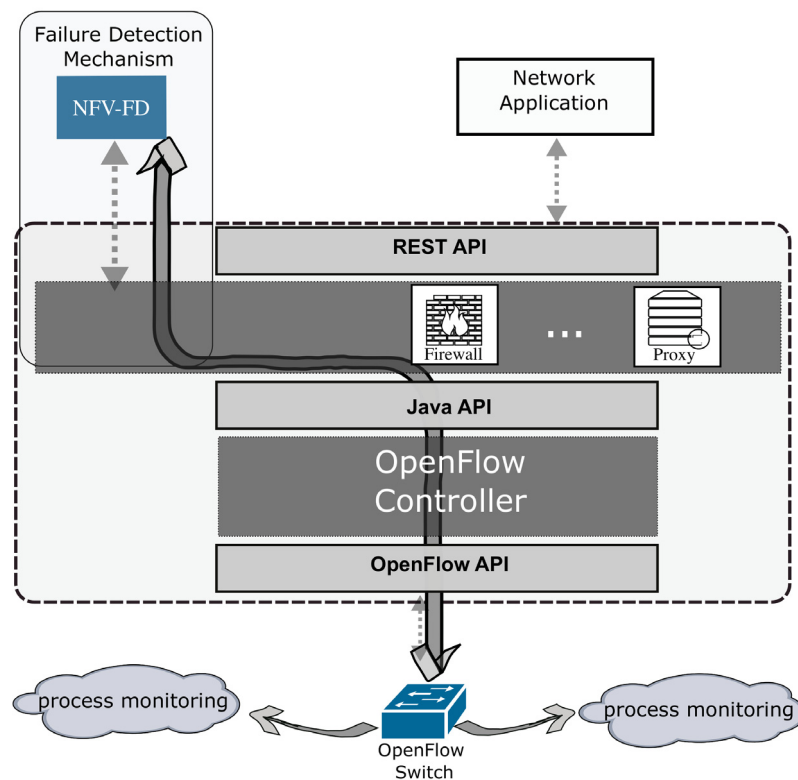


Figura 6.2: NFV-FD: arquitetura.

O NFV-FD também detecta falhas nos enlaces de comunicação. Para tanto, o detector utiliza um mecanismo para descoberta de topologia, chamado LLDP (*Link Layer Discovery Protocol*). Os próprios switches enviam pacotes do tipo LLDP para informar qualquer tipo de

mudança na topologia da rede. Neste caso, o FMod captura, filtra e processa todos os pacotes do tipo LLDP, enviando então as informações relevantes ao NFV-FD. Por exemplo, a mensagem *port s2-eth2 changed: DOWN* indica que a porta *eth2* do *switch s2* está inativa. O NFV-FD por sua vez verifica se existem processos conectados nesta porta e os marca como inacessíveis (*unreachable*).

No trabalho, o NFV-FD é comparado com a alternativa em que o detector de falhas é implementado no próprio controlador, e não como uma VNF isolada. Os resultados experimentais mostram que, apesar de o tempo de detecção ser levemente maior (aproximadamente 120ms), existe um impacto menor no controlador SDN – o uso de CPU diminui. Portanto, a abordagem baseada em NFV permite aos controladores executarem somente as suas tarefas nativas, consequentemente atendendo a uma maior taxa de requisições.

6.2.2 Consenso como um Serviço NFV-COIN: VNF-Consensus

Por padrão, o plano de controle em uma rede SDN é centralizado, o que por definição pode levar a problemas em termos de disponibilidade, escalabilidade e desempenho (Canini et al., 2015). Enquanto que um plano de controle distribuído pode resolver estes problemas, é essencial garantir uma sincronização consistente entre os múltiplos controladores SDN, o que não é uma tarefa trivial. Em geral, as soluções existentes implementam as tarefas de sincronização no próprio controlador, enquanto que outras soluções sincronizam o plano de dados ao invés de sincronizar o plano de controle. Em (Venâncio et al., 2021) o serviço NFV-COIN VNF-Consensus implementa o algoritmo de consenso Paxos, de forma a manter consistente um plano de controle distribuído de uma rede SDN. O serviço VNF-Consensus não necessita de nenhuma modificação no plano de dados nem no protocolo SDN.

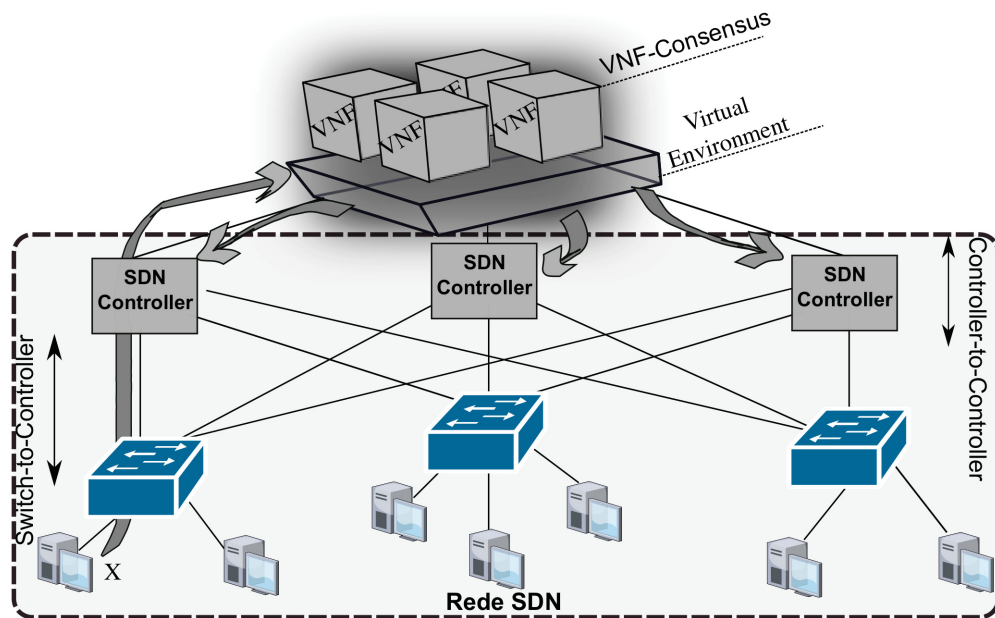


Figura 6.3: VNF-Consensus: arquitetura.

A Figura 6.3 mostra o funcionamento da VNF-Consensus. Para garantir a consistência do plano de controle distribuído, antes que uma nova regra OpenFlow seja instalada em um *switch* no plano de dados, essa regra deve ser replicada para todos os controladores SDN. O controlador correspondente ao *switch* que recebeu a regra e a envia para a VNF-Consensus, de forma que ela execute o algoritmo Paxos e garanta que todos os controladores concordem

sobre a regra a ser instalada. Após a execução do consenso, o plano de dados pode ser então atualizado de maneira consistente com a nova regra. Dessa forma, se algum dos controladores falha, qualquer outro controlador SDN do plano de controle pode assumir o controlador falho, uma vez que eles garantidamente possuem a mesma visão da rede.

Resultados experimentais mostram que a VNF-Consensus é mais eficiente do que executar o consenso nos próprios controladores para garantir a consistência de um plano de controle distribuído. Ao usar a abordagem baseada em NFV, o uso de CPU do controlador diminui de 62.1% para aproximadamente 34.4%. Portanto, o controlador consegue lidar com uma maior quantidade de requisições por segundo: um aumento de aproximadamente 53%. Ao mesmo tempo, a latência para instalar uma regra de maneira consistente diminui consideravelmente: até 18.5% em comparação com a solução alternativa. Por fim, ao utilizar a VNF-Consensus, a vazão é aproximadamente 3.6 vezes maior ao considerar cenários com alta demanda.

6.2.3 Difusão Confiável e Ordenada como um Serviço NFV-COIN: NFV-RBCast

Tradicionalmente, os usuários finais utilizam serviços de difusão confiável ou implementam os algoritmos na própria aplicação distribuída, ou, alternativamente, instalam/mantêm *middlewares* com essa funcionalidade. A solução NFV-RBCast (Venâncio et al., 2019) é proposta no contexto NFV-COIN, permitindo que a própria rede forneça serviços nativos de difusão confiável e ordenada de mensagens. A NFV-RBCast oferece diversos serviços de difusão, sendo eles: difusão confiável, difusão atômica, difusão FIFO, difusão atômica FIFO e difusão atômica causal. As aplicações que necessitam utilizar o serviço podem acessar as primitivas através de uma API chamada RBCast.

O serviço NFV-RBCast garante a ordem em que as mensagens são entregues (*i.e.*, *delivered*) através de um sequenciador de mensagens, implementado como uma VNF, denominado de VNF-Sequencer (ilustrado na Figura 6.4). Seguindo o funcionamento padrão do protocolo OpenFlow, sempre que um *switch* recebe um pacote que não possui uma entrada correspondente na sua tabela de fluxos, o pacote deve ser encaminhado para o controlador utilizando um pacote do tipo *packet_in*. Se o pacote *packet_in* indica que o fluxo foi originado de um serviço NFV-RBCast, o controlador instala uma regra no *switch* para que todos os próximos pacotes deste fluxo sejam enviados diretamente para a VNF-Sequencer.

Ao receber estes pacotes, a VNF-Sequencer executa o algoritmo de ordenação especificado no início da comunicação com o serviço. A VNF-Sequencer então insere um número de sequência dentro de cada pacote, que é posteriormente enviado para o grupo de processos destinatários. Ao receber uma mensagem originada de uma difusão, cada processo entrega a mensagem para a aplicação final de acordo com o número de sequência inserido pela VNF-Sequencer.

Resultados experimentais mostram que a VNF-Consensus consegue atingir uma vazão de até 15000 mensagens por segundo para a difusão atômica, considerando 50 processos participantes. Em comparação com a difusão confiável, a difusão atômica aumenta a latência da entrega de cada mensagem em aproximadamente 32%, o que é um valor esperado, visto que a difusão atômica garante a ordem total das mensagens.

6.3 A ARQUITETURA NFV-COIN

Esta seção tem como objetivo propor uma arquitetura totalmente compatível com a arquitetura de referência NFV-MANO para suportar, de maneira transparente, a oferta e execução de serviços NFV-COIN arbitrários. É importante notar que existem dois papéis de usuários que manipulam os serviços de maneiras distintas. O primeiro papel é do operador da rede, responsável

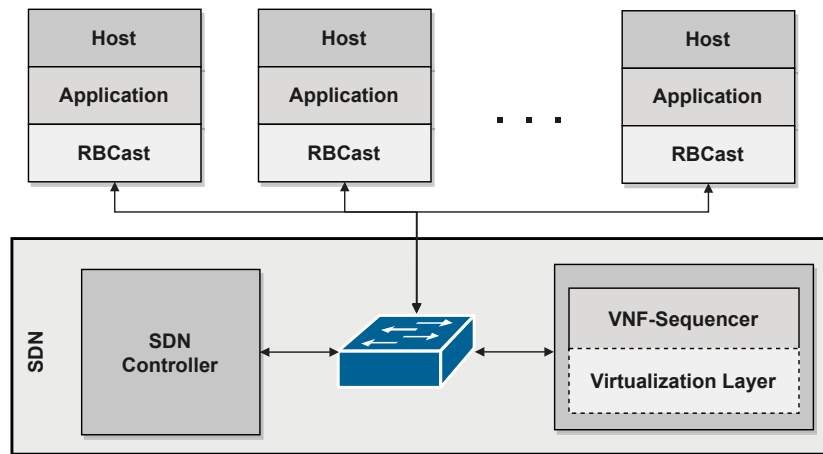


Figura 6.4: NfV-RBCast: arquitetura.

pela gerência usual do NFV-MANO. O operador da rede pode, por exemplo, registrar novos serviços no NFV-MANO através do VNFD para que eles possam ser posteriormente instanciados. Em geral, o operador da rede realiza operações relacionadas somente ao NFV-MANO. O segundo papel é do usuário final, responsável pelo uso propriamente dito dos serviços. O usuário final é quem se beneficia diretamente da arquitetura NFV-COIN, solicitando a instanciação de novos serviços e utilizando em sua aplicação as funcionalidades de tais serviços.

A Figura 6.5 ilustra a proposta da arquitetura NFV-COIN. Como pode ser observado na figura, a arquitetura inclui um Módulo de Usuário (*User Module*), executado no servidor do usuário final. Este módulo oferece uma API para que as aplicações dos usuários acessem e gerenciem os serviços NFV-COIN. Essa API é denominada de NFV-COIN *User API*, e pode ser implementada como uma biblioteca contendo um conjunto de primitivas que permite o acesso e gerenciamento dos serviços NFV-COIN. A API também permite que o usuário solicite a criação, configuração, uso, monitoramento e remoção de qualquer tipo de serviço. A NFV-COIN *User API* também possibilita a integração de múltiplos serviços nas suas aplicações locais de maneira transparente e unificada.

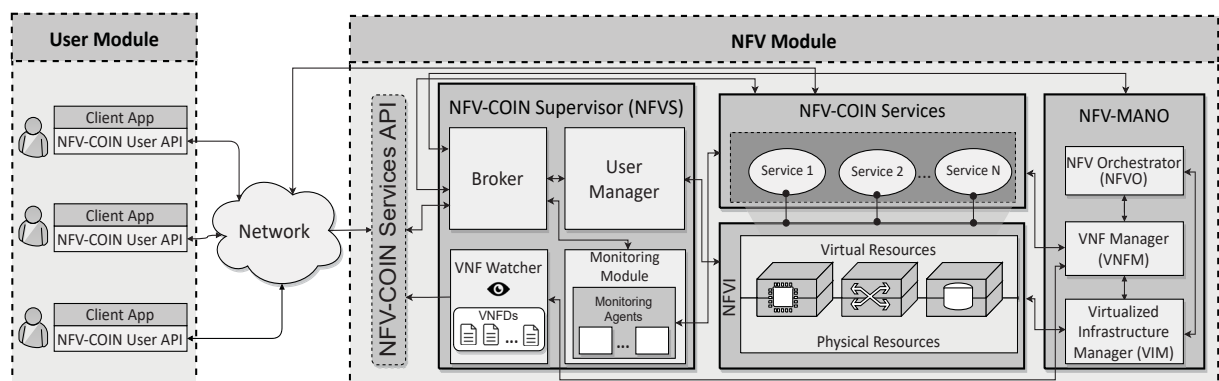


Figura 6.5: A arquitetura NFV-COIN.

A Figura 6.5 ilustra o módulo NFV (*NFV Module*), executado na própria rede e consiste de um sistema tradicional NFV (*i.e.*, componentes NFV-MANO), além de um supervisor, denominado de NFVS (*NFV-COIN Supervisor*). O NFVS pode ser visto como um *gateway* para os serviços NFV-COIN, composto de diversos módulos que executam tarefas distintas (todos os módulos serão descritos em detalhes adiante).

Além disso, a arquitetura implementa outra API, denominada de *NFV-COIN Services API*. Essa API é responsável por receber todas as requisições das aplicações e acionar o componente correspondente à requisição. A *NFV-COIN Services API* pode ser vista como ponto de entrada na rede para todos os tipos de requisições que necessitem manipular os serviços NFV-COIN, tais como a solicitação para criação de um novo serviço, alteração de parâmetros, execução de funcionalidades específicas, entre outros. Portanto, o *User Module* comunica-se com os serviços NFV-COIN através da *NFV-COIN User API (client-side)* e a *NFV-COIN Services API (server-side)*. A seguir, a *NFV-COIN Services API* e o NFVS são descritos em detalhes.

6.3.1 NFV-COIN Services API

A *NFV-COIN Services API* é responsável por manter todos os conjuntos de primitivas referentes a cada serviço NFV-COIN. Por exemplo, considere o serviço de detecção de falhas NFV-FD (descrito na Seção 6.2.1). Uma das primitivas que este serviço deve oferecer é a geração de uma lista de processos suspeitos de falha. Por sua vez, a *NFV-COIN User API* precisa de uma primitiva para iniciar essa funcionalidade. Outros exemplos de primitivas para o serviço NFV-FD incluem a inclusão/remoção de processos a serem monitorados. Por outro lado, o serviço VNF-Consensus deve incluir primitivas para propor e decidir valores. A *NFV-RBCast* por sua vez deve definir primitivas para os diferentes tipos de difusão suportados, incluindo funções para enviar e entregar valores para a aplicação. A Tabela 6.2 especifica alguns exemplos das primitivas para cada serviço descrito acima.

Tabela 6.2: Exemplos de primitivas da *NFV-COIN Services API*.

Serviço	Método	URL	Descrição
NFV-FD	POST	/nfv-fd/include/{process_id}	Inclui um processo para ser monitorado
NFV-FD	POST	/nfv-fd/remove/{process_id}	Remove um processo do monitoramento
NFV-FD	GET	/nfv-fd/list/	Lista todos os processos suspeitos de falha
VNF-Consensus	POST	/vnf-consensus/set_role/{process_id}/{role}	Define o papel de um processo. Opções: <i>proposer</i> , <i>acceptor</i> , <i>learner</i>
VNF-Consensus	POST	/vnf-consensus/propose/{value}	Envia um valor para o consenso
VNF-Consensus	GET	/vnf-consensus/learn/{round}	Recebe um valor decidido em uma rodada
NFV-RBCast	POST	/nfv-rbcast/send/reliable/{data}	Envia um dado por difusão confiável
NFV-RBCast	POST	/nfv-rbcast/send/atomic/{data}	Envia um dado por difusão atômica
NFV-RBCast	POST	/nfv-rbcast/send/fifo/{data}	Envia um dado por difusão FIFO
NFV-RBCast	POST	/nfv-rbcast/send/atomic/fifo/{data}	Envia um dado por difusão atômica FIFO
NFV-RBCast	POST	/nfv-rbcast/send/atomic/causal/{data}	Envia um dado por difusão atômica causal
NFV-RBCast	GET	/nfv-rbcast/recv	Recebe um valor enviado por difusão
NFV-RBCast	GET	/nfv-rbcast/deliver	Entrega um valor para a aplicação

Na tabela, a coluna *Serviço* representa o nome do serviço NFV-COIN. A coluna *Método* representa o tipo de ação a ser executada pela primitiva. Em geral, os métodos mais comuns são: *POST*, utilizado para representar uma primitiva que insere ou envia dados; *GET*, utilizado normalmente para obter uma informação do serviço e; *DELETE*, utilizado para remover algum dado referente ao serviço. Por sua vez, a coluna *URL (Uniform Resource Locator)* indica o endereço exato para acessar a primitiva do serviço especificado. Por fim, a última

coluna *Descrição* apresenta uma breve descrição da funcionalidade oferecida pela primitiva correspondente.

Além disso, devemos considerar que um ambiente NFV é dinâmico, *i.e.*, novas VNFs e serviços podem ser incluídos de maneira dinâmica de acordo com o operador da rede. Portanto, a NFV-COIN *Services API* também fornece uma primitiva, denominada neste trabalho de *discovery*. Tal serviço tem como objetivo retornar ao usuário um catálogo atualizado, contendo os serviços e primitivas disponíveis na rede. Para tanto, sempre que um novo serviço for incluído no catálogo através do NFV-MANO, é de responsabilidade do NFVS (em particular, do VNF *Watcher*, descrito adiante) identificar essa alteração e atualizar a primitiva *discovery* para obter as novas primitivas deste serviço.

A NFV-COIN *Services API* permite, portanto, que as aplicações dos usuários finais acessem os serviços NFV-COIN através da NFV-COIN *User API*. Ambas as APIs poderiam ser implementadas, por exemplo, através de uma interface REST (*REpresentational State Transfer*), facilitando a interoperabilidade e integração entre aplicações heterogêneas e sistemas de diferentes desenvolvedores.

Como visto nos exemplos acima, a NFV-COIN *Services API* consiste de um conjunto de primitivas que são definidas caso a caso para cada serviço NFV-COIN, *i.e.*, não existe um conjunto genérico de primitivas que podem ser adotados por todos os serviços. Portanto, é tarefa do desenvolvedor da função especificar todas as primitivas que a VNF oferece. Esta especificação pode ser feita, por exemplo, através de uma extensão no VNFD da VNF.

A NFV-COIN *Services API* deve ser genérica o suficiente para: (i) permitir que as aplicações dos usuários finais acessem os serviços NFV-COIN na rede através de chamadas padronizadas; (ii) suportar diferentes tipos de tecnologias de virtualização, como NFV, SDN e computação em nuvem e; (iii) ser compatível com a maioria das linguagens de programação, facilitando a interoperabilidade entre VNFs de diferentes desenvolvedores.

6.3.2 NFVS: NFV-COIN *Supervisor*

O *NFV Module* consiste dos componentes originais da arquitetura NFV-MANO (*i.e.*, NFVO, VNFM, VIM), além da infraestrutura física onde os serviços são virtualizados e executados (*i.e.*, a NFVI). De forma a permitir a execução e gerência de serviços virtualizados em uma arquitetura NFV-COIN, um novo módulo deve ser incorporado na arquitetura NFV-MANO, denominado neste capítulo de NFVS. O NFVS é o componente principal da arquitetura NFV-COIN, e suas funcionalidades podem ser organizadas em quatro módulos: o *Broker*, o *User Manager*, o VNF *Watcher* e o *Monitoring Module*. Cada módulo é descrito a seguir.

O NFVS atua como um *gateway* entre a aplicação do usuário final e os serviços NFV-COIN utilizados pela aplicação. Dessa forma, todas as requisições feitas pela aplicação através da NFV-COIN *User API* são recebidas pela NFV-COIN *Services API*, que por sua vez encaminha a requisição ao *Broker* do NFVS. O *Broker* é, portanto, o módulo responsável por analisar e processar as requisições. Após o processamento, o *Broker* determina para qual módulo a requisição deverá ser encaminhada. Caso seja uma requisição de manipulação e/ou monitoramento do serviço, o *Broker* envia a requisição diretamente ao serviço NFV-COIN; caso seja uma requisição de gerência, o *Broker* envia a requisição ao NFV-MANO.

O NFVS também possui o módulo *User Manager*, responsável pela configuração específica das sessões de cada usuário. Essa configuração deve garantir inclusive que a comunicação entre a aplicação e o usuário seja segura. Usuários devem ser autorizados e toda a comunicação com os serviços NFV-COIN deve ser autenticada e a confidencialidade deve ser garantida. Além disso, o *User Manager* automaticamente cria redes virtuais dedicadas para cada usuário, permitindo que as instâncias dos serviços NFV-COIN de cada usuário sejam executados

de maneira individual. Essa abordagem garante isolamento, privacidade e segurança; além disso, não há competição por recursos, prevenindo gargalos que poderiam surgir ao passo em que o número de usuários aumenta na rede. O *User Manager* também mantém, em um banco de dados local, informações sobre os usuários e as suas respectivas conexões com a rede NFV, incluindo, por exemplo, quais serviços o usuário está utilizando, qual rede virtual está associada a qual usuário, quantas conexões estão ativas/inativas, entre outros.

O terceiro módulo presente no NFVS é o *VNF Watcher*. Este módulo é responsável pelo monitoramento periódico de novos serviços cadastrados. No momento em que um novo serviço é registrado no NFV-MANO pelo operador da rede, o componente *VNF Watcher* detecta esta mudança e realiza o processamento do VNFD correspondente. Neste momento, o *VNF Watcher* verifica se o VNFD (descrito em detalhes na Seção 2.2) foi descrito de acordo com as especificações NFV-COIN (*i.e.*, incluindo as primitivas de execução e monitoramento). Em seguida, o *VNF Watcher* extrai as informações relevantes do VNFD e atualiza a NFV-COIN *Services API* com as primitivas do novo serviço. Somente após esta validação o serviço poderá ser instanciado pelo usuário final.

Por fim, o NFVS também possui um módulo de monitoramento (*Monitoring Module*), para monitorar o estado dos serviços instanciados, além de fornecer informações sobre o próprio sistema, incluindo a NFVI e os canais de comunicação. Enquanto o NFV-MANO realiza somente o monitoramento de métricas específicas do dispositivo virtual (*e.g.*, uso de CPU, memória, rede), a ideia principal do *Monitoring Module* é a possibilidade de oferecer um monitoramento customizado sobre cada serviço NFV-COIN. Portanto, o desenvolvedor da VNF além de especificar as primitivas que permitem o seu acesso de maneira genérica, também pode-se especificar um conjunto de primitivas que permitam o seu monitoramento. Considerando, por exemplo, o serviço NFV-FD, uma possível métrica de monitoramento customizada é a verificação de quais processos estão marcados como suspeitos de falha.

6.4 FUNCIONALIDADES DA ARQUITETURA NFV-COIN

A seguir são descritas as operações fundamentais da arquitetura NFV-COIN para permitir que serviços NFV-COIN sejam oferecidos, gerenciados e utilizados: (i) operações para a criação de conexões seguras e autenticadas entre os usuários finais e os serviços NFV-COIN; (ii) operações relacionadas a comunicação de dados entre usuário final e os serviços NFV-COIN e; (iii) operações referentes ao gerenciamento de alto nível dos serviços NFV-COIN.

6.4.1 Estabelecimento de Conexão com Serviços NFV-COIN

Antes de solicitar a criação de uma nova instância de um serviço NFV-COIN, o usuário deve verificar, através da primitiva *discovery*, se o serviço requisitado está presente e disponível na rede. Após verificar que o serviço está disponível e pode ser criado, uma requisição para criação do serviço é feita através da NFV-COIN *User API*. Essa requisição é recebida pela NFV-COIN *Services API*, que por sua vez a encaminha ao *Broker*. Em seguida, o *Broker* envia uma mensagem ao módulo *User Manager* para estabelecer uma conexão segura entre o usuário e o serviço. Em particular, uma nova sessão deve ser criada entre a aplicação e o serviço.

Para que uma nova sessão seja criada, o *User Manager* realiza a autenticação de ambas as partes: cliente *i.e.*, aplicação e o novo serviço. Após a autenticação, o *User Manager* realiza uma requisição ao sistema NFV-MANO para criar uma nova rede virtual para o usuário, caso ela ainda não exista. Por fim, o *Broker* registra a sessão do usuário no banco de dados local do *User Manager*. Após a sessão ser criada, o *Broker* realiza uma requisição (através do NFV-MANO)

para instanciar o serviço NFV-COIN. O NFV-MANO por sua vez instancia o serviço solicitado na rede virtual do usuário. O *Broker* aguarda até que a instanciação termine, para então retornar uma mensagem para a aplicação informando que o serviço está operacional.

6.4.2 Comunicação com os Serviços NFV-COIN

Após o estabelecimento da conexão, a aplicação do usuário final pode se comunicar com o serviço NFV-COIN de duas formas: comunicação direta e dedicada com o serviço ou através de requisições pela própria NFV-COIN *Services API*. A escolha de qual método utilizar depende somente das restrições de desempenho impostas pela comunicação de dados. Apesar da comunicação direta e dedicada ser mais eficiente, a opção de comunicação pela NFV-COIN *Services API* pode ser mais conveniente, uma vez que a interface REST oferece uma forma simplificada de acessar funcionalidades através das primitivas.

A comunicação direta entre usuário final e serviço pode ser a melhor escolha para situações em que um grande volume de dados precisa ser transferido. Neste caso, a comunicação direta é inicialmente estabelecida através do *Broker*, que realiza as configurações necessárias de roteamento entre a aplicação e a rede virtual onde o serviço NFV-COIN está instanciado. Além disso, o *Broker* cria e retorna ao usuário uma conexão dedicada com o serviço, por exemplo, via *sockets* em uma conexão TCP. Após essa etapa, a aplicação pode transferir diretamente dados para o serviço NFV-COIN.

6.4.3 Gerenciando Serviços NFV-COIN

A NFV-COIN *Services API* fornece um conjunto de operações para o gerenciamento alto nível dos serviços NFV-COIN. Exemplos incluem as operações para criação, remoção, reconfiguração e *auto-scaling* – operações tradicionais do NFV-MANO. Enquanto estas operações envolvem os módulos da arquitetura NFV-MANO, outras operações de gerenciamento podem ser executadas diretamente no serviço virtualizado. Em especial, o gerenciamento dos serviços NFV-COIN deve permitir operações customizadas, tais como a atualização *on-the-fly* de parâmetros específicos do serviço. Durante o processamento da requisição, o *Broker* consegue identificar para qual componente a requisição deverá ser encaminhada.

É importante observar que o *Broker* é responsável pelos serviços NFV-COIN como um todo, e não pelas VNFs que compõe o serviço, que são de responsabilidade do próprio NFV-MANO. Portanto, o *Broker* não é responsável pelo ciclo de vida das VNFs/SFCs correspondentes, mas sim pelo gerenciamento dos serviços NFV-COIN.

6.5 IMPLEMENTAÇÃO E RESULTADOS EXPERIMENTAIS

Como prova de conceito, um protótipo da arquitetura NFV-COIN foi desenvolvido. O protótipo implementa todos os componentes descritos na Seção 6.3 utilizando a linguagem Python 3.9. A implementação considera serviços baseados em *containers* da plataforma *Docker*. Para a implementação das APIs foram utilizadas as bibliotecas Python *Flask*¹ e *Eve*².

Para permitir que a arquitetura NFV-COIN identifique as funcionalidades (*i.e.*, primitivas) de um serviço de rede, é necessário que o desenvolvedor do serviço especifique estas mudanças no descritor da função (*i.e.*, VNFD). Essa descrição também deve ser feita para as métricas customizadas de monitoramento. O exemplo a seguir mostra um VNFD sintetizado para um serviço NFV-COIN de difusão confiável.

¹<https://flask.palletsprojects.com/en/2.2.x/>

²<https://docs.python-eve.org/en/stable/>

```

1 topology_template:
2   node_templates:
3     capabilities:
4       nfv_compute:
5         properties:
6           mem_size: 512 MB
7           num_cpus: 1
8     properties:
9       type: container
10      image: ubuntu-rbcast
11    primitives:
12      - send_reliable:
13        url: /nfv-rbcast/send/reliable
14        data:
15          - value
16      - send_atomic:
17        url: /nfv-rbcast/send/atomic
18        data:
19          - value
20      - recv:
21        url: /nfv-rbcast/recv
22        data: null
23    monitoring:
24      - failed_processes:
25        url: /nfv-rbcast/monitoring/failed_process
26        data: null
27      - correct_processes:
28        url: /nfv-rbcast/monitoring/correct_process
29        data: null
30      - messages_sent:
31        url: /nfv-rbcast/monitoring/messages_sent
32        data:
33          - process_id

```

No exemplo acima é possível observar que este serviço oferece três primitivas: enviar valor por difusão confiável (linha 12); enviar valor por difusão atômica (linha 16) e; receber valor (linha 20). O serviço oferece também três métricas de monitoramento customizado: obter lista de processos falhos (linha 24); obter lista de processos corretos (linha 27) e; exibir quantidade de mensagens enviadas de um determinado processo (linha 30). Para cada primitiva e métrica de monitoramento, dois atributos são atribuídos: (i) *URL*: determina a URL correspondente utilizada para acionar a primitiva/monitoramento e; (ii) *data*: parâmetro que especifica quais valores devem ser enviados junto à requisição.

Para a NFV-COIN *User API*, uma biblioteca foi implementada contendo as funções principais para gerenciar e manipular os serviços NFV-COIN. As principais funções desta biblioteca são descritas a seguir:

- *discovery()*: retorna uma lista dos serviços NFV-COIN disponíveis na rede.
- *get_primitives(service_name)*: retorna uma lista das primitivas oferecidas pelo serviço NFV-COIN especificado.
- *create_service(service_name)*: solicita a instanciação do serviço especificado. Caso seja o primeiro serviço do usuário, uma nova sessão é estabelecida pelo componente *User Manager*, autenticando o usuário e criando uma rede isolada para a criação dos

serviços. Para este protótipo, a sessão consiste dos seguintes atributos: identificador de usuário; quota (disco, memória e CPU) e; endereço IP da rede isolada.

- *execute(service_name, primitive, data)*: especifica um serviço NFV-COIN e uma primitiva para ser executada.

A seguir, a arquitetura NFV-COIN é avaliada. Para tanto, dois serviços foram implementados no conceito NFV-COIN: um serviço de cache e um serviço de balanceamento de cargas, descritos em detalhes nas próximas subseções. A avaliação foi realizada em uma máquina com processador Intel Core I7 @ 2.5GHz, 16GB RAM DDR4, uma placa de rede Ethernet de 1 Gbps e Linux Ubuntu 16.04. Todos os resultados são apresentados com um intervalo de confiança de 95%.

6.5.1 Avaliação do Impacto na Latência e Vazão

Antes de avaliar a arquitetura através dos serviços implementados, é importante determinar se existem impactos significativos na latência e vazão ao utilizar a arquitetura NFV-COIN. Para avaliar tais métricas, um serviço simples de encaminhamento de pacotes (*forwarder*) foi utilizado – o objetivo é que o processamento da função não interfira nestes resultados. Para os experimentos desta subseção, dois cenários são comparados. O primeiro, avalia o serviço *forwarder* sem o suporte da NFV-COIN, *i.e.*, o usuário envia tráfego diretamente para a função. No segundo cenário, o serviço *forwarder* é criado através da arquitetura NFV-COIN.

O primeiro experimento, detalhado na Tabela 6.3, compara os níveis de vazão (em Mbps) para três casos: *Baseline* (vazão sem NFV-COIN), Conexão Dedicada (com NFV-COIN) e Conexão via NFV-COIN *Services API* (com NFV-COIN).

Tabela 6.3: Comparação da vazão considerando diferentes cenários.

Vazão (Mbps)	Mínima	Máxima	Média	Desvio Padrão
<i>Baseline</i>	905	999	967	25
NFV-COIN: Conexão Dedicada	902	963	936	15
NFV-COIN: Conexão via API	36	163	130	1

Naturalmente, a conexão via API apresenta um impacto significativo na vazão, uma vez que existe uma sobrecarga em cada pacote. Essa sobrecarga consiste principalmente das requisições REST das APIs (NFV-COIN *User API* e NFV-COIN *Services API*). No entanto, como mencionado na seção anterior, tal conexão é utilizada somente para primitivas de configuração e manipulação e, portanto, não devem afetar o desempenho global do serviço. Por outro lado, a conexão dedicada apresenta resultados muito mais próximos ao *baseline*: existe uma redução na vazão de aproximadamente 3,2%, em média e em comparação com a conexão via API é, na média, aproximadamente 7 vezes maior.

O segundo experimento, detalhado na Tabela 6.4, compara para os mesmos casos o impacto na latência. A latência foi medida para cada pacote, e consiste no tempo desde o envio até o recebimento do pacote pelo destino. A tabela exibe os resultados para um envio de 1000 pacotes.

Assim como no experimento anterior, os níveis de latência entre o *baseline* e a conexão dedicada são próximos: um aumento de aproximadamente 0,011ms, em média, para cada pacote. Como esperado, a latência da conexão via API apresenta valores maiores para a latência, também decorrentes da comunicação adicional entre APIs existente entre a aplicação e o destino.

Tabela 6.4: Comparação da latência: *baseline* e NFV-COIN.

Latência (ms)	Mínima	Máxima	Média	Desvio Padrão
<i>Baseline</i>	0.053	0.074	0.056	0.002
<i>NFV-COIN: Conexão Dedicada</i>	0.058	0.082	0.067	0.001
<i>NFV-COIN: Conexão via API</i>	4.594	8.768	5.127	0.375

É importante observar, no entanto, que a arquitetura NFV-COIN não introduz perdas significativas de desempenho. Se um serviço exige alta vazão e baixa latência, a conexão dedicada da NFV-COIN pode ser aplicada sem comprometer o desempenho do serviço.

6.5.2 Avaliação do Serviço de Cache

Um serviço para cache Web foi implementado, responsável por armazenar requisições HTTP e as respostas correspondentes. Para gerar uma nova entrada na cache, uma chave é gerada da seguinte forma: $key = hash(K) \% N$, onde K é a requisição e N é a quantidade máxima de entradas do tipo chave-valor na cache. A resposta da requisição HTTP é posteriormente armazenada na tabela *hash* como $H[key] = response$. Para a substituição de entradas na cache, este serviço implementa a política LRU (*Least Recently Used*).

Este serviço NFV-COIN de cache implementa as seguintes primitivas: (i) *generate_key*: gera uma chave baseada em uma requisição; (ii) *put*: insere uma nova entrada na cache; (iii) *get* busca uma entrada na cache. Pode retornar um valor caso esteja presente ou retorna uma falha de cache (*i.e.*, *cache miss*); (iv) *remove* remove uma entrada específica da cache e; (v) *clear* remove todas as entradas da cache.

Para o experimento descrito a seguir, dois cenários são comparados: execução do serviço com e sem o suporte da arquitetura NFV-COIN. No cenário sem o suporte NFV-COIN, o serviço é executado e configurado manualmente através do NFV-MANO.

O experimento ilustrado na Figura 6.6 avalia o serviço NFV-COIN de cache em termos de latência. Para ambos os cenários, compara-se o tempo total para a Requisição 1 – requisição inicial, não possui valor correspondente na cache, *e.g.*, *cache miss* e; Requisição 2 – cache já possui o valor correspondente da requisição. Para cada caso, 1000 requisições foram executadas, sendo as barras a média do tempo total da requisição e as barras de erro o desvio padrão.

Como neste experimento a Requisição 1 resulta necessariamente em um *cache miss*, é possível dividir o tempo total da requisição em três primitivas executadas pelo serviço: tempo para geração da chave, tempo para obter o recurso solicitado e o tempo para inserir a nova entrada na cache. Naturalmente, a primitiva que consome a maior parte do tempo é para obter o novo recurso (GET), representando aproximadamente 92,8% do tempo total da requisição para ambos os cenários. Ao comparar as duas abordagens, o cenário com NFV-COIN apresenta uma latência total aproximadamente 11,4% maior. Isso ocorre devido aos passos extras de comunicação entre aplicação, NFV-COIN *User API* e NFV-COIN *Services API*.

No caso da Requisição 2, como o valor solicitado já está presente em cache, o tempo total da requisição não possui o tempo para inserir uma nova entrada na cache. Dessa forma, o tempo total da requisição diminui de 375ms para 12,2ms para o cenário sem NFV-COIN, e de 418ms para 14,5ms para o cenário com NFV-COIN, representando uma redução de aproximadamente 96% no tempo total da requisição para ambos os cenários. Neste caso, a diferença de latência total entre cenários é significativamente menor: aproximadamente 2,3ms de diferença.

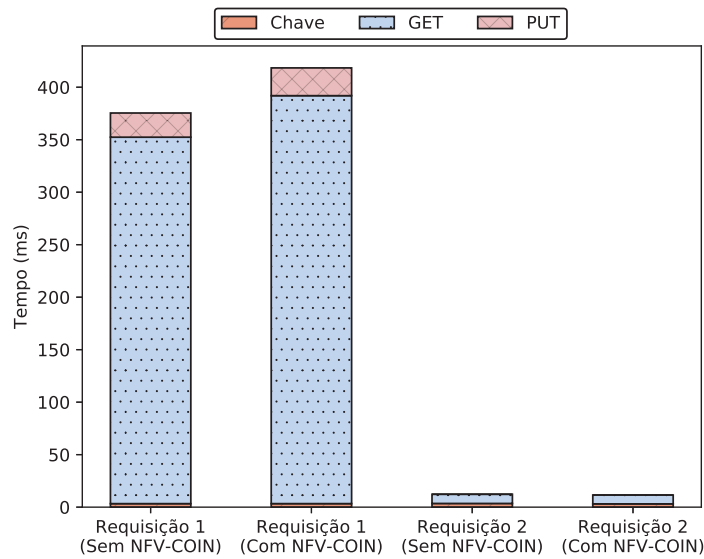


Figura 6.6: Avaliação da latência para o serviço de cache.

6.5.3 Avaliação do Serviço de Balanceamento de Cargas

O segundo serviço implementado como NFV-COIN é um balanceador de cargas. O serviço utiliza uma política baseada em *round-robin* de forma que cada requisição é enviada sequencialmente a um conjunto de N servidores. Para este serviço, existem duas formas possíveis de comunicação (conforme descrito na Seção 6.4.2): comunicação através da NFV-COIN *Services* API e comunicação direta dedicada. Neste caso, a comunicação através da API é utilizada para configurar parâmetros específicos da função, enquanto que a comunicação dedicada é utilizada para o envio propriamente dito dos dados.

O serviço de balanceamento de cargas implementa as seguintes primitivas: (i) *set_servers*: configura o balanceador de cargas com um conjunto de N servidores; (ii) *add_server*: adiciona um servidor no balanceador de cargas; (iii) *remove_server*: remove um servidor do balanceamento de cargas e; (iv) *set_policy*: configura a política de distribuição de tráfego.

O experimento ilustrado na Figura 6.7 avalia a vazão para o serviço de balanceamento de cargas. Assim como no experimento anterior, dois cenários são comparados: com ou sem o suporte da arquitetura NFV-COIN. Para o cenário com NFV-COIN, a conexão dedicada descrita na Seção 6.4.2 é utilizada. Ou seja, as requisições são enviadas diretamente da aplicação para o serviço NFV-COIN, *i.e.*, não passam pela NFV-COIN *Services* API. Já para o cenário sem NFV-COIN, o serviço foi implementado de forma adjacente à aplicação do usuário. Para ambos os cenários, compara-se a vazão conforme a quantidade de servidores do serviço varia. O experimento foi executado durante uma hora para cada caso e as barras exibem a média da vazão (em requisições por segundo) e as barras de erro o desvio padrão.

É possível observar que, para cada servidor adicionado ao balanceador de cargas, a vazão aumenta significativamente. Para ambos os cenários, o aumento na vazão foi semelhante: aproximadamente 2x mais ao variar de um para dois servidores; e 1,4x mais ao variar de dois a três e de três a quatro servidores.

No entanto, é importante observar as diferenças entre o cenário com e sem NFV-COIN. Independente da quantidade de servidores, a arquitetura NFV-COIN introduz uma redução na vazão de somente 5%, causada pelo passo adicional de comunicação com o ambiente NFV.

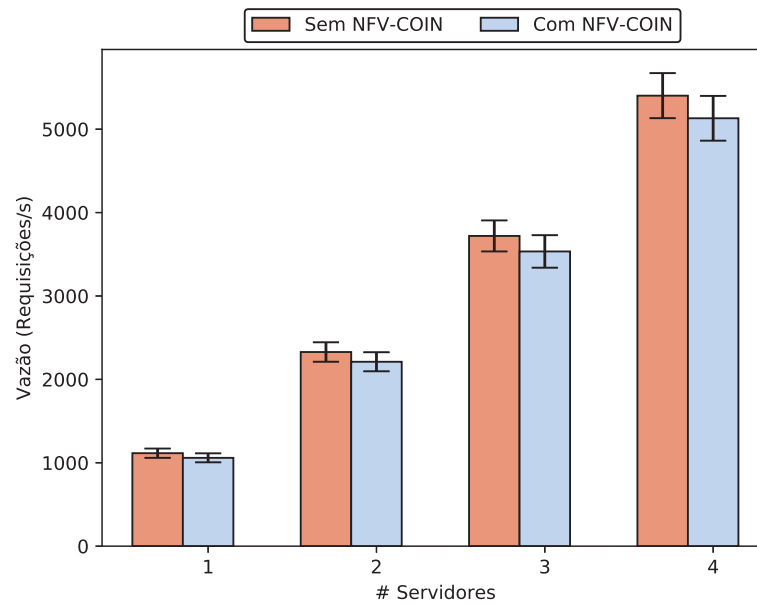


Figura 6.7: Avaliação da vazão para o serviço de balanceamento de cargas.

6.6 CONCLUSÕES PARCIAIS

Este capítulo explora uma alternativa para o oferecimento e execução de serviços arbitrários dentro da própria rede utilizando tecnologia NFV. Em contraste com os serviços virtualizados tradicionais, os serviços NFV-COIN são solicitados pelas aplicações de usuários finais através de primitivas fornecidas por uma API. Três serviços propostos no contexto NFV-COIN foram descritos como estudos de caso e claramente mostram as vantagens de executar serviços COIN utilizando tecnologia NFV.

Para permitir que tais serviços sejam oferecidos e executados de maneira transparente e unificada, uma arquitetura NFV-COIN é proposta como extensão à arquitetura de referência NFV-MANO. A arquitetura é composta de quatro componentes principais: (i) uma API para a aplicação que permite à aplicação solicitar e utilizar serviços NFV-COIN; (ii) a NFV-COIN *Services* API, executada na própria rede e oferece uma interface para requisições aos serviços; (iii) o NFVS, componente responsável por intermediar as requisições entre aplicação e serviços NFV-COIN e; (iv) os próprios serviços NFV-COIN.

É importante observar que o objetivo da arquitetura NFV-COIN não é oferecer ganhos significativos de desempenho, mas manter níveis similares de desempenho enquanto permite a simplicidade e flexibilidade na execução de serviços de rede arbitrários.

7 CONCLUSÃO

Nesta Tese de Doutorado foram apresentadas contribuições em dois contextos diferentes: soluções de tolerância a falhas de serviços virtualizados e a oferta e gerenciamento de serviços inovadores na rede.

A primeira contribuição da Tese descreve o NHAM, uma arquitetura NFV que garante a alta disponibilidade de serviços virtualizados *stateful* e *stateless*. Para tanto, o NHAM possui um gerenciador de estados, denominado de *State Synchronizer*, responsável por capturar periodicamente o estado interno de cada VNF através de *checkpoints*. Além disso, o NHAM possibilita a definição de VNFs redundantes através de quatro mecanismos diferentes, escolhidos de acordo com os requisitos de cada função. Para garantir a recuperação correta e consistente de ponta-a-ponta de SFCs, o NHAM também define uma estratégia que combina as técnicas de *Checkpoint/Restore* com o gerenciamento de *buffers* de uma SFC, permitindo a recuperação mesmo após a falha simultânea de múltiplas VNFs. Sob um ponto de vista prático, o NHAM consegue oferecer um conjunto completo de funcionalidades que permite tornar os serviços virtualizados altamente disponíveis sem a necessidade de alterações no código-fonte das VNFs. Diferente das demais soluções, a arquitetura proposta é totalmente compatível e integrada à arquitetura de referência NFV-MANO, facilitando a sua integração com outros sistemas e plataformas NFV.

Um protótipo foi implementado e resultados experimentais avaliam o impacto gerado pelo NHAM nas SFCs. Em especial, os mecanismos baseados no método *Active-Active* possuem menor tempo de recuperação. Além disso, o tempo de recuperação não aumenta conforme o tamanho da SFC aumenta, mostrando a escalabilidade da solução. Para os casos em que uma SFC apresenta múltiplas falhas simultâneas, o tempo de recuperação não é linear, comprovando a eficácia da solução. Já para o impacto na vazão, o mecanismo que apresentou menor degradação foi o 1R-AA, já que ele opera diretamente em memória. Por outro lado, o mecanismo MR-AA apesar de apresentar o menor tempo de recuperação, possui o maior impacto na vazão do serviço, consequência da sincronização do grupo de réplicas. No entanto, este mecanismo atinge a métrica *five nines* de disponibilidade para todos os cenários. Portanto, com os resultados apresentados na avaliação experimental, é possível afirmar que o NHAM é uma solução eficaz para melhorar a robustez dos serviços virtualizados, podendo atingir níveis de disponibilidade similares aos de sistemas comerciais de telecomunicações.

A segunda contribuição desta Tese descreve a FIT-SFC, uma arquitetura que utiliza estratégias de replicação para suportar serviços virtualizados seguros e altamente disponíveis. A FIT-SFC utiliza $3f + 1$ réplicas de cada componente interno para tolerar até f falhas bizantinas. Além disso, cada réplica possui um componente *Voter*, responsável por autorizar o processamento de um determinado pacote somente após receber um quórum de $2f + 1$ mensagens idênticas. Ainda, a FIT-SFC emprega estratégias para tolerar falhas para SFCs que processam tanto tráfego *stateful* ou *stateless*, evitando a perda ou duplicação de pacotes em casos de falha. Diferente das demais soluções da literatura, a arquitetura FIT-SFC pode ser vista como a primeira que (i) é totalmente compatível com a arquitetura de referência IETF de SFCs; (ii) considera falhas não somente das VNFs, mas de qualquer componente interno da arquitetura SFC e; (iii) que tolera não somente falhas por parada, como também falhas por omissão e falhas bizantinas.

Um protótipo da arquitetura foi desenvolvido, além de três serviços de rede: um DNS, um DNS autenticado e um balanceador de cargas. Os resultados obtidos demonstram que a latência para o serviço DNS e DNS autenticado é aproximadamente 4,7 e 3,6 vezes maior,

respectivamente. Já para o balanceador de cargas, a latência é 7,7 vezes maior – um impacto maior gerado principalmente pela execução do consenso. Naturalmente, existe um custo em termos de latência, uma vez que os componentes são replicados e consequentemente a quantidade de pacotes redundantes transitando na rede aumenta, além dos recursos computacionais para manter as réplicas. No entanto, tal custo é justificável para serviços que executam funcionalidades críticas.

Finalmente, a terceira contribuição desta Tese investiga a união entre o paradigma NFV e o conceito COIN, denominado de NFV-COIN. Uma arquitetura NFV-COIN é proposta para superar um problema desafiador: a oferta de serviços inovadores nativamente na própria rede. Na prática, a arquitetura NFV-COIN permite a oferta, execução e gerenciamento de serviços virtualizados arbitrários de maneira unificada e padronizada. Para tanto, a arquitetura conta com duas APIs, uma para a aplicação solicitar requisições para os serviços NFV-COIN, e outra para a rede, atuando como uma interface para o recebimento destas requisições. Além disso, o componente principal da arquitetura NFV-COIN é o NFVS, responsável por intermediar toda a comunicação entre aplicação e serviços NFV-COIN. É importante observar que a arquitetura NFV-COIN foi projetada de maneira totalmente compatível com a arquitetura de referência NFV-MANO, permitindo a sua integração com diversos ambientes NFV.

Para avaliar a proposta, um protótipo da arquitetura NFV-COIN foi implementado, assim como um serviço de cache e um serviço de balanceamento de cargas. Resultados experimentais avaliam a arquitetura em termos de latência e vazão, demonstrando que a solução proposta não insere prejuízos significativos. Independente do serviço em execução, a arquitetura NFV-COIN introduz uma redução na vazão de aproximadamente 3,2%. Para o serviço de cache, observou-se uma latência aproximadamente 11,4% maior em casos de cache *miss*, devido aos passos extras de comunicação das APIs. Já para os casos de cache *hit*, a diferença é consideravelmente menor – aproximadamente 2,3ms. Por fim, para o serviço de balanceamento de cargas, a arquitetura introduziu uma redução na vazão de somente 5%. Tais resultados demonstram que a arquitetura NFV-COIN consegue manter níveis similares de desempenho dos serviços, ao passo em que simplifica e flexibiliza a oferta e execução dos mesmos.

Analizando as contribuições desta Tese em um contexto geral, é possível concluir que as soluções propostas resolvem os problemas definidos no Capítulo 1. Em particular, as duas primeiras contribuições (*i.e.*, NHAM e FIT-SFC) apresentam avanços significativos para o estado da arte no que diz respeito à confiabilidade de serviços virtualizados baseados em NFV. Após um exaustivo exame dos trabalhos relacionados da literatura, estas soluções são as primeiras que resolvem, de maneira completa, o problema de alta disponibilidade para NFV considerando, inclusive, os padrões propostos pela ETSI e IETF. O objetivo é que, no futuro, tais soluções possam ser implantadas e amplamente adotadas em ambientes de produção. Já no contexto da terceira contribuição (*i.e.*, NFV-COIN), com a implantação de redes 5G e já com o desenvolvimento de redes 6G, os requisitos de baixa latência, alta vazão e programabilidade da rede podem ser facilitados explorando os conceitos de NFV-COIN.

Os trabalhos futuros no contexto do NHAM incluem ampliar os tipos de virtualização suportados. Em particular, permitir o suporte para máquinas virtuais, considerando inclusive o uso de *unikernels*. O suporte a outras tecnologias de virtualização, tais como o KVM (*Kernel-based Virtual Machine*) (Kivity et al., 2007) e o QEMU (*Quick EMUlator*) (Bellard, 2005), também pode contribuir para a posicionar a solução em um contexto mais amplo. Também será investigada a possibilidade de detectar e recuperar novos tipos de falha, como, por exemplo, falhas no sistema operacional, nos componentes do NFV-MANO e nos próprios componentes do NHAM. A definição de estratégias baseadas em aprendizado de máquinas no contexto de prevenção e previsão de falhas é outro caminho que poderá ser investigado. Outro trabalho futuro

reside na possibilidade de implementar mecanismos customizados e específicos para preservar o estado interno de dispositivos virtuais baseados em NFV. Apesar de que utilizar ferramentas de *Checkpoint/Restore* disponibilizadas pela indústria/academia facilitam a implementação da arquitetura NHAM, incorporar uma solução própria específica para o contexto NFV pode levar a ganhos significativos de desempenho. Por fim, o NFV-MANO permite o ajuste automático de recursos das VNFs de acordo com a demanda (*i.e.*, *autoscaling*). Se uma determinada VNF está sobrecarregada, o NFV-MANO pode criar de maneira dinâmica uma nova instância desta VNF para auxiliar com o processamento do tráfego. No entanto, este processo não considera que uma VNF pode ser *stateful*. Dessa forma, propor o NHAM neste contexto é outro caminho promissor e desafiador.

Os trabalhos futuros no contexto da FIT-SFC incluem a definição de estratégias para otimizar o tempo total de processamento e diminuir a sobrecarga da solução em termos de latência e vazão. Para tanto, está prevista a execução do tráfego em lotes. Tal estratégia irá, possivelmente, aumentar significativamente a vazão dos serviços. Além disso, está prevista a otimização do algoritmo de consenso utilizado, responsável por grande parte da sobrecarga em casos onde a SFC é do tipo *stateful* global. Também será investigada a definição e avaliação da FIT-SFC em um ambiente distribuído e federado. Uma vez que SFCs podem executar entre domínios administrativos distintos (Huff et al., 2020), é essencial que a FIT-SFC possa ser configurada para tais cenários, permitindo, por exemplo, que cada domínio possua um SC e um SFF correspondente.

Já no contexto da NFV-COIN, trabalhos futuros incluem investigar, implementar e avaliar diversos tipos de serviços NFV-COIN em diferentes contextos. Considerando principalmente o aumento nas aplicações que utilizam computação de borda (*edge processing*) e *edge-to-cloud continuum*, não há dúvidas de que a utilização de tecnologias de virtualização para a oferta e execução de serviços de rede arbitrários é um caminho promissor que deve continuar a ser investigado. Além disso, a integração de diferentes tecnologias de virtualização invariavelmente introduzem riscos de segurança. Portanto, outro trabalho futuro é utilizar as vantagens oferecidas pelo conceito NFV-COIN para o desenvolvimento de soluções voltadas à confiabilidade e segurança de sistemas, tanto através do desenvolvimento de serviços NFV-COIN, quanto através do suporte da própria arquitetura. Ainda, apesar de que o desempenho é sempre um desafio para serviços baseados em virtualização, investigar e desenvolver serviços NFV-COIN de alto desempenho é um desafio complexo, mas que pode trazer diversos avanços ao estado da arte. Por fim, uma vez que COIN é um conceito que está em constante evolução, adaptar e expandir a arquitetura NFV-COIN para atender os novos desafios que possam surgir no futuro é outra direção de pesquisa.

REFERÊNCIAS

- Al-Rubaye, S., Rodriguez, J., Al-Dulaimi, A., Mumtaz, S. e Rodrigues, J. J. (2019). Enabling digital grid for industrial revolution: self-healing cyber resilient platform. *IEEE Network*, 33(5):219–225.
- Avizienis, A., Laprie, J.-C., Randell, B. e Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.
- Ban, B. et al. (2002). Jgroups, a toolkit for reliable multicast communication.
- Bellard, F. (2005). Qemu, a fast and portable dynamic translator. Em *USENIX annual technical conference, FREENIX Track*, volume 41, página 46. Califórnia, USA.
- Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.
- 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.
- Borran, F., Hutle, M., Santos, N. e Schiper, A. (2012). Quantitative analysis of consensus algorithms. *IEEE Transactions on Dependable and Secure Computing*, 9(2).
- Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G. et al. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.
- Bressana, P., Zilberman, N., Vucinic, D. e Soulé, R. (2020). Trading latency for compute in the network. Em *Proceedings of the 2020 Workshop on Network Application Integration/CoDesign, NAI@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*, páginas 35–40. ACM.
- Cachin, C., Guerraoui, R. e Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition.
- Canini, M., Kuznetsov, P., Levin, D. e Schmid, S. (2015). A distributed and robust SDN control plane for transactional network updates. Em *IEEE Conference on Computer Communications (INFOCOM)*.
- Cappanera, P., Paganelli, F. e Paradiso, F. (2019). Vnf placement for service chaining in a distributed cloud environment with multiple stakeholders. *Computer Communications*, 133:24–40.
- Castro, M. e Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461.
- Castro, M., Liskov, B. et al. (1999). Practical byzantine fault tolerance. Em *OsDI*, volume 99, páginas 173–186.

- Chandra, T. D., Hadzilacos, V. e Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722.
- Chandra, T. D. e Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2).
- Charron-Bost, B., Pedone, F. e Schiper, A. (2010). Replication: Theory and practice, ser. Incs, vol. 5959.
- Chiosi, M., Clarke, D., Willis, P., Reid, A., Feger, J., Bugenhagen, M., Khan, W., Fargano, M., Cui, C., Deng, H. et al. (2012). Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. Em *SDN and OpenFlow World Congress*, páginas 22–24.
- Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. e Warfield, A. (2005). Live migration of virtual machines. Em *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, páginas 273–286.
- Cotroneo, D., De Simone, L., Iannillo, A., Lanzaro, A., Natella, R., Fan, J. e Ping, W. (2014a). Network function virtualization: Challenges and directions for reliability assurance. Em *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, páginas 37–42. IEEE.
- Cotroneo, D., De Simone, L., Iannillo, A. K., Lanzaro, A., Natella, R., Fan, J. e Ping, W. (2014b). Network function virtualization: Challenges and directions for reliability assurance. Em *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, páginas 37–42. IEEE.
- Cotroneo, D., De Simone, L., Liguori, P., Natella, R. e Bidokhti, N. (2019). How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. Em *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, páginas 200–211.
- CRIU (2019). Checkpoint/Restore In Userspace.
- Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N. e Warfield, A. (2008). Remus: High availability via asynchronous virtual machine replication. Em *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, páginas 161–174. San Francisco.
- Dang, H. T., Bressana, P., Wang, H., Lee, K. S., Zilberman, N., Weatherspoon, H., Canini, M., Pedone, F. e Soulé, R. (2020). P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*.
- Dang, H. T., Canini, M., Pedone, F. e Soulé, R. (2016). Paxos made switch-y. *ACM SIGCOMM Computer Communication Review*, 46(2):18–24.
- Défago, X., Schiper, A. e Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.
- Docker (2017). Docker - build, ship, and run any app, anywhere. <https://www.docker.com/>. Acessado em novembro de 2017.

- Elnozahy, E. N., Alvisi, L., Wang, Y.-M. e Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408.
- Esposito, F., Mushtaq, M., Berno, M., Davoli, G., Borsatti, D., Cerroni, W. e Rossi, M. (2020). Necklace: An architecture for distributed and robust service function chains with guarantees. *IEEE Transactions on Network and Service Management*, 18(1):152–166.
- ETSI (2019). Open Source MANO.
- ETSI (Available at <http://www.etsi.org/technologies-clusters/technologies/nfv>, Accessed on October 02, 2016). Etsi gs nfv 002: Architectural framework.
- ETSI, N. (2014). Network functions virtualisation (nfv); management and orchestration. *NFV-MAN*, 1:v0.
- ETSI, N. (2015a). Network functions virtualisation (nfv) ecosystem; report on sdn usage in nfv architectural framework. *ETSI GS NFV-EVE*, 5:v1.
- ETSI, N. (2015b). Network functions virtualization (nfv) infrastructure overview. *NFV-INF*, 1:V1.
- Fan, J., Guan, C., Zhao, Y. e Qiao, C. (2017). Availability-aware mapping of service function chains. Em *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, páginas 1–9. IEEE.
- Fan, J., Jiang, M., Rottenstreich, O., Zhao, Y., Guan, T., Ramesh, R., Das, S. e Qiao, C. (2018). A framework for provisioning availability of nfv in data center networks. *IEEE Journal on Selected Areas in Communications*, 36(10):2246–2259.
- Fischer, M. J., Lynch, N. A. e Paterson, M. S. (1985a). Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382.
- Fischer, M. J., Lynch, N. A. e Paterson, M. S. (1985b). Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2).
- Garcia, V., Huff, A., Santos, C. R. e P. Duarte Jr, E. (2020). Network service topology: Formalization, taxonomy and the custom specification model. *International Conference on Computer Networks*, TBT(TBT):1–53.
- Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S. e Akella, A. (2014). Opennf: Enabling innovation in network function control. Em *ACM SIGCOMM Computer Communication Review*, páginas 163–174. ACM.
- Ghaznavi, M., Jalalpour, E., Wong, B., Boutaba, R. e Mashtizadeh, A. (2020). Fault tolerant service function chaining. Em *SIGCOMM*, páginas 198–210, Online. ACM.
- Gill, P., Jain, N. e Nagappan, N. (2011). Understanding network failures in data centers: measurement, analysis, and implications. Em *Proceedings of the ACM SIGCOMM 2011 Conference*, páginas 350–361.
- Gonzalez, A. J., Nencioni, G., Kamisiński, A., Helvik, B. E. e Heegaard, P. E. (2018). Dependability of the nfv orchestrator: State of the art and research challenges. *IEEE Communications Surveys & Tutorials*, 20(4):3307–3329.

- Graham, R. L., Woodall, T. S. e Squyres, J. M. (2005). Open mpi: A flexible high performance mpi. Em *International Conference on Parallel Processing and Applied Mathematics*, páginas 228–239. Springer.
- Gray, J. e Siewiorek, D. P. (1991). High-availability computer systems. *Computer*, 24(9):39–48.
- Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V. et al. (2014). What bugs live in the cloud? a study of 3000+ issues in cloud systems. Em *Proceedings of the ACM symposium on cloud computing*, páginas 1–14.
- Gunawi, H. S., Hao, M., Suminto, R. O., Laksono, A., Satria, A. D., Adityatama, J. e Eliazar, K. J. (2016). Why does the cloud stop computing? lessons from hundreds of service outages. Em *Proceedings of the Seventh ACM Symposium on Cloud Computing*, páginas 1–16.
- Hadzilacos, V. e Toueg, S. (1993). Fault-tolerant broadcasts and related problems. Em *Distributed systems (2nd Ed.)*, páginas 97–145. ACM.
- Halpern, J. e Pignataro, C. (2015). Service Function Chaining (SFC) Architecture. RFC 7665, IETF.
- Han, B., Gopalakrishnan, V., Ji, L. e Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97.
- Han, B., Gopalakrishnan, V., Kathirvel, G. e Shaikh, A. (2017). On the resiliency of virtual network functions. *IEEE Communications Magazine*, 55(7):152–157.
- Harchol, Y., Hay, D. e Orenstein, T. (2018). Ftnvf: Fault tolerant virtual network functions. Em *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, páginas 141–147.
- Herker, S., An, X., Kiess, W., Beker, S. e Kirstaedter, A. (2015). Data-center architecture impacts on virtualized network functions service chain embedding with high availability requirements. Em *2015 IEEE Globecom Workshops (GC Wkshps)*, páginas 1–7. IEEE.
- Huff, A. et al. (2018). A holistic approach to define service chains using click-on-osv on different nfv platforms. Em *2018 IEEE Global Communications Conference (GLOBECOM)*, páginas 1–6. IEEE.
- Huff, A. et al. (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.
- Hunt, P., Konar, M., Junqueira, F. P. e Reed, B. (2010). {ZooKeeper}: Wait-free coordination for internet-scale systems. Em *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.
- Jepsen, T., Lerner, A., Pedone, F., Soulé, R. e Cudré-Mauroux, P. (2021). In-network support for transaction triaging. *Proceedings of the VLDB Endowment*, 14(9):1626–1639.
- Kablan, M., Alsudais, A., Keller, E. e Le, F. (2017). Stateless network functions: Breaking the tight coupling of state and processing. Em *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, páginas 97–112.

- Khalid, J. e Akella, A. (2019). Correctness and performance for stateful chained network functions. Em *The 16th NSDI*, páginas 501–516, Boston. USENIX Association.
- Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A. e Akella, A. (2016). Paving the way for {NFV}: Simplifying middlebox modifications using {StateAlyzr}. Em *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, páginas 239–253.
- Kivity, A., Kamay, Y., Laor, D., Lublin, U. e Liguori, A. (2007). kvm: the linux virtual machine monitor. Em *Proceedings of the Linux symposium*, volume 1, páginas 225–230. Dttawa, Dntorio, Canada.
- Kohler, E., Morris, R., Chen, B., Jannotti, J. e Kaashoek, M. F. (2000). The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297.
- Kong, J. et al. (2017a). Guaranteed-availability network function virtualization with network protection and vnf replication. Em *Global Communications Conference*, páginas 1–6.
- Kong, J., Kim, I., Wang, X., Zhang, Q., Cankaya, H. C., Xie, W., Ikeuchi, T. e Jue, J. P. (2017b). Guaranteed-availability network function virtualization with network protection and vnf replication. Em *GLOBECOM 2017-2017 IEEE Global Communications Conference*, páginas 1–6. IEEE.
- Koo, R. e Toueg, S. (1987). Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, 25(1):23–31.
- Kshemkalyani, A. D. e Singhal, M. (2011). *Distributed computing: principles, algorithms, and systems*. Cambridge University Press.
- Kulkarni, S. G., Liu, G., Ramakrishnan, K., Arumaithurai, M., Wood, T. e Fu, X. (2018). Reinforce: Achieving efficient failure resiliency for network function virtualization based services. Em *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, páginas 41–53. ACM.
- Lac, C., Adams, R. e et al (2017). Network Function Virtualisation (NFV); Reliability; Report on the resilience of NFV-MANO critical capabilities. GR NFV-REL 007 V1.1.1. Relatório técnico, ETSI.
- Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2).
- Lamport, L. (2001). Paxos Made Simple. *SIGACT News*, 32(4):51–58.
- Lamport, L. (2006). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125.
- Lamport, L. (2019). Time, clocks, and the ordering of events in a distributed system. Em *Concurrency: the Works of Leslie Lamport*, páginas 179–196. ACM.
- Lamport, L., Shostak, R. e Pease, M. (2019). The byzantine generals problem. Em *Concurrency: the works of leslie lamport*, páginas 203–226. ACM.
- Lao, C., Le, Y., Mahajan, K., Chen, Y., Wu, W., Akella, A. e Swift, M. (2021). ATP: In-network aggregation for multi-tenant learning. Em *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, páginas 741–761.

- Li, J., Liang, W., Huang, M. e Jia, X. (2020). Reliability-aware network service provisioning in mobile edge-cloud networks. *IEEE Transactions on Parallel and Distributed Systems*, 31(7):1545–1558.
- Li, S. e Crandall, J. (2017). TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0. OASIS committee specification draft, Open Standards for the Information Society. <http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html>.
- Li, W., Kanso, A. e Gherbi, A. (2015). Leveraging linux containers to achieve high availability for cloud services. Em *2015 IEEE International Conference on Cloud Engineering*, páginas 76–83. IEEE.
- Li, Y. e Chen, M. (2015). Software-defined network function virtualization: A survey. *IEEE Access*, 3:2542–2553.
- Liu, M., Luo, L., Nelson, J., Ceze, L., Krishnamurthy, A. e Atreya, K. (2017). Incbricks: Toward in-network computation with an in-network cache. Em *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, páginas 795–809.
- Marques, J. A., Luizelli, M. C., Tavares da Costa Filho, R. I. e Gaspar, L. P. (2019). An optimization-based approach for efficient network monitoring using in-band network telemetry. *Journal of Internet Services and Applications*, 10(1):1–20.
- McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S. e Turner, J. (2008). Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74.
- Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.
- 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.
- Misa, C., Durairajan, R., Rejaie, R. e Willinger, W. (2021). Revisiting network telemetry in coin: A case for runtime programmability. *IEEE Network*, 35(5):14–20.
- Mockapetris, P. V. (1987). Rfc1035: Domain names-implementation and specification.
- Mullender, S. et al. (1993). *Distributed systems*, volume 12. acm press United States of America.
- Nakamura, H., Adams, R. e et al (2016). Network Functions Virtualisation (NFV); Reliability; Report on Models and Features for End-to-End Reliability. GS NFV-REL 003 V1.1.1. Relatório técnico, ETSI.
- Ongaro, D. e Ousterhout, J. (2014). In search of an understandable consensus algorithm. Em *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, páginas 305–319.
- OpenStack (2019). OpenStack - open source software for creating private and public clouds.

- Pattaranantakul, M., He, R., Song, Q., Zhang, Z. e Meddahi, A. (2018). Nfv security survey: From use case driven threat analysis to state-of-the-art countermeasures. *IEEE Communications Surveys & Tutorials*, 20(4):3330–3368.
- Piedad, F. e Hawkins, M. (2001). *High availability: design, techniques, and processes*. Prentice Hall Professional.
- Pradhan, D. K. (1996). *Fault-tolerant computer system design*. Prentice-Hall, Inc.
- QEMU (2022). Qemu micro checkpointing. <https://wiki.qemu.org/Features/MicroCheckpointing>. Acessado em novembro de 2022.
- Quinn, P. et al. (2018). Network Service Header (NSH) - RFC 8300. Relatório técnico, Internet Engineering Task Force.
- Quittek, J., Bauskar, P., BenMeriem, T., Bennett, A., Besson, M. e et al (2014). Network Functions Virtualisation (NFV); Management and Orchestration. GS NFV-MAN 001 V1.1.1. Relatório técnico, ETSI.
- Rajagopalan, S., Williams, D. e Jamjoom, H. (2013a). Pico replication: A high availability framework for middleboxes. Em *Proceedings of the 4th annual Symposium on Cloud Computing*, páginas 1–15, New York. ACM.
- Rajagopalan, S., Williams, D., Jamjoom, H. e Warfield, A. (2013b). Split/merge: System support for elastic execution in virtual middleboxes. Em *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, páginas 227–240.
- Randell, B., Lee, P. e Treleaven, P. C. (1978). Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165.
- Sahoo, J., Mohapatra, S. e Lath, R. (2010a). Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. Em *2010 Second International Conference on Computer and Network Technology*, páginas 222–226. IEEE.
- Sahoo, S. K., Criswell, J. e Adve, V. (2010b). An empirical study of reported bugs in server software with implications for automated bug diagnosis. Em *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, páginas 485–494. IEEE.
- 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.
- Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.
- Schöller, M., Khan, N. e et al (2015). Network Function Virtualisation (NFV); Resiliency Requirements. GS NFV-REL 001 V1.1.1. Relatório técnico, ETSI.
- Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K. e Shi, G. (2012). Design and implementation of a consolidated middlebox architecture. Em *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, páginas 323–336.
- Sharma, S., Engelmann, A., Jukan, A. e Gumaste, A. (2020). Vnf availability and sfc sizing model for service provider networks. *IEEE Access*, 8:119768–119784.

- Sherry, J., Gao, P. X., Basu, S., Panda, A., Krishnamurthy, A., Maciocco, C., Manesh, M., Martins, J., Ratnasamy, S., Rizzo, L. et al. (2015). Rollback-recovery for middleboxes. Em *ACM SIGCOMM Computer Communication Review*, páginas 227–240. ACM.
- Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S. e Sekar, V. (2012). Making middleboxes someone else's problem: Network processing as a cloud service. Em *Proceedings of the ACM SIGCOMM 2012*.
- Tavares, T. N. et al. (2018). Niep: Nfv infrastructure emulation platform. Em *International Conference on Advanced Information Networking and Applications*, páginas 173–180.
- Tokusashi, Y., Dang, H. T., Pedone, F., Soulé, R. e Zilberman, N. (2019). The case for in-network computing on demand. Em *Proceedings of the Fourteenth EuroSys Conference 2019*, páginas 1–16.
- Turchetti, R. et al. (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. et al. (2017). Nfv-fd: Implementation of a failure detector using network virtualization technology. *International Journal of Network Management*, 27(6):e1988.
- Turchetti, R. C. e Duarte, E. P. (2015). Implementation of failure detector based on network function virtualization. *Network, iEEE*, páginas 19–25.
- Venâncio, G., Turchetti, R. C., Camargo, E. T. e Duarte Jr, E. P. (2021). 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-rbcst: Enabling the network to offer reliable and ordered broadcast services. Em *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, páginas 1–10. IEEE.
- Wang, C., Chen, X., Jia, W., Li, B., Qiu, H., Zhao, S. e Cui, H. (2018). PLOVER: Fast, multi-core scalable virtual machine fault-tolerance. Em *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, páginas 483–489.
- Wang, L. et al. (2021). Ddqp: A double deep q-learning approach to online fault-tolerant sfc placement. *IEEE Transactions on Network and Service Management*, 18(1):118–132.
- Woo, S., Sherry, J., Han, S., Moon, S., Ratnasamy, S. e Shenker, S. (2018). Elastic scaling of stateful network functions. Em *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, páginas 299–312.
- Zeng, D., Ansari, N., Montpetit, M.-J., Schooler, E. M. e Tarchi, D. (2021). Guest editorial: In-network computing: Emerging trends for the edge-cloud continuum. *IEEE Network*, 35(5):12–13.

APÊNDICE A – PUBLICAÇÕES

A.1 TRABALHOS PUBLICADOS NO ÂMBITO DA TESE

1. **Venâncio, G.** e Duarte, E. P. (2022). NHAM: An NFV High Availability Architecture for Building Fault-Tolerant Stateful Virtual Functions and Services. *In XI Latin-American Symposium on Dependable Computing (LADC'2022). Best Paper Award.*
2. **Venâncio, G.**, Turchetti, R. C., e Duarte, E. P. (2022). NFV-COIN: Unleashing The Power of In-Network Computing with Virtualization Technologies. *In Journal of Internet Services and Applications (JISA).*
3. Garcia, V., **Venâncio, G.** e Duarte, E. P. (2022). Arquitetura e Mapeamento de Serviços Virtualizados de Rede Tolerantes a Falhas e Intrusão. Em XXIII Workshop de Testes e Tolerância a Falhas (WTF'2022).
4. **Venâncio, G.** e Duarte, E. P. (2021). Uma Arquitetura de Alta Disponibilidade para Serviços Virtualizados de Rede. Em XXII Workshop de Testes e Tolerância a Falhas (WTF'2021).
5. **Venâncio, G.**, Turchetti, R. C., e Duarte, E. P. (2021). NFVinc: Disponibilizando Serviços Inovadores na Rede com Tecnologia NFV. Em XIX Workshop de Computação em Clouds e Aplicações (WCGA'2021).
6. **Venâncio, G.** e Duarte, E. P. (2020). Uma Arquitetura de Alta Disponibilidade para Funções Virtualizadas de Rede. Em XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2020).

A.2 TRABALHOS PUBLICADOS NO ÂMBITO DO GRUPO DE PESQUISA

1. Flauzino, J., Garcia, V., Huff, A., **Venâncio, G.** e Duarte, E. P. (2021). Gerência e Orquestração de Funções e Serviços de Rede Virtualizados em Nuvem CloudStack. Em XXVI Workshop de Gerência e Operação de Redes e Serviços (WGRS'2021).
2. Huff, A., **Venâncio, G.**, Garcia, V. e Duarte, E. P. (2020). Building Multi-domain Service Function Chains Based on Multiple NFV Orchestrators. *In 6th IEEE Conference on Network Functions Virtualization and Software Defined Networking (NFV-SDN'2020).*
3. Flauzino, J., Garcia, V., **Venâncio, G.** e Duarte, E. P. (2020). Além do OpenStack: Disponibilizando o Suporte para Funções Virtualizadas de Rede NFV-MANO no CloudStack. Em XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2020).
4. **Venâncio, G.**, Turchetti, R. C., de Camargo, E. T. e Duarte, E. P. (2020). VNF-Consensus: A Virtual Network Function for Maintaining a Consistent Distributed SDN Control Plane. *In International Journal of Network Management (IJNM).*
5. **Venâncio, G.**, Turchetti, R. C. e Duarte, E. P. (2019). NFV-RBCast: Enabling the Network to Offer Reliable and Ordered Broadcast Services. *In IX Latin-American Symposium on Dependable Computing (LADC'2019).*

6. **Venâncio, G.**, Turchetti, R. C., de Camargo, E. T. e Duarte, E. P. (2019). VNF-Consensus: A Virtual Network Function for Maintaining a Consistent Distributed SDN Control Plane. In *9th Latin American Network Operations and Management Symposium (LANOMS'2019)*.
7. Huff, A., **Venâncio, G.** e Duarte, E. P. (2019). Multi-SFC: Orquestração de SFCs Distribuídas sobre Múltiplas Nuvens em Múltiplos Domínios com Múltiplas Plataformas NFV. Em *XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2019)*.
8. Garcia, V., **Venâncio, G.**, Tavares, T., Marcuzzo, L., Franco, M., Bondan, L., Duarte, E. P., Santos, C., Granville, L., Schaeffer-Filho, A., Turck, F. (2019). On the Design and Development of Emulation Platforms for NFV-based Infrastructures. In *International Journal of Grid and Utility Computing (IJGUC)*.
9. Garcia, V. F., Marcuzzo, L., **Venâncio, G.**, Bondan, L., Nobre, J., Schaeffer-Filho, A., Santos, C., Granville, L. Z. e Duarte, E. P. (2019). An NSH-Enabled Architecture for Virtualized Network Function Platforms. In *International Conference on Advanced Information Networking and Applications (AINA'2019)*.
10. Bondan, L., Franco, M., Marcuzzo, L., **Venâncio, G.**, Santos, R. L., Pfitscher, R. J., Scheid, E. J., Stiller, B., De Turck, F., Duarte, E. P., Schaeffer-Filho, A. E., Santos, C.; Granville, L. Z. (2019) FENDE: Marketplace-Based Distribution, Execution, and Life Cycle Management of VNFs. In *IEEE Communications Magazine*, vol. 57, no. 1, pp. 13-19.
11. **Venâncio, G.**, Garcia, V., Marcuzzo, L., Tavares, T., Franco, M., Bondan, L., Schaeffer-Filho, A., Santos, C., Granville, L. e Duarte, E. P. (2019). Beyond VNFM: Filling the Gaps of the ETSI VNF Manager to Fully Support VNF Lifecycle Operations. In *International Journal of Network Management (IJNM)*.