Resource Management for Efficient, Scalable and Resilient Network Function Chains

Dissertation
for the award of the degree

Doctor of Philosophy (Ph.D.)
Division of Mathematics and Natural Sciences
of the Georg-August-Universität Göttingen

within the PhD Programme in Computer Science (PCS)
Georg-August University School of Science (GAUSS)

Submitted by
Sameer G. Kulkarni
from Dharwad, Karnataka, India

Göttingen
June 2018

Thesis Committee:	Prof. Dr. Xiaoming Fu, Georg-August-Universität Göttingen
	Prof. Dr. K. K. Ramakrishnan, University of California, Riverside, USA
	PD. Dr. Mayutan Arumaithurai, Georg-August-Universität Göttingen
Examination Board:	
Reviewer:	Prof. Dr. Xiaoming Fu, Georg-August-Universität Göttingen
Other Reviewers:	Prof. Dr. Klaus Wehrle, RWTH Aachen
	Prof. Dr. Dieter Hogrefe, Georg-August-Universität Göttingen
Further Members of the Examination Board:	Prof. Dr. K. K. Ramakrishnan, University of California, Riverside, USA
	Prof. Dr. Jens Grabowski, Georg-August-Universität Göttingen
	PD. Dr. Mayutan Arumaithurai, Georg-August-Universität Göttingen

Date of Oral Examination: 04 July 2018

Abstract

Networks, the basis of the modern connected world, have evolved beyond the connectivity services. *Network Function*s (NFs) or traditionally the middleboxes are the basis of realizing different types of services such as security, optimization functions, and value added services. Typically, multiple NFs are chained together (also known as *Service Function Chaining*) to realize distinct network services, which are pivotal in providing the policy enforcement and performance in networks. *Network Function Virtualization* (NFV) is becoming more prevalent and enabling the softwarized NFs to fast replace the traditional dedicated hardware based middleboxes in *Communication Service Provider* (CSP) networks. However, *Virtualized Network Function* (VNF) chains posit several systems and network level resource management and failure resiliency challenges: to ensure optimal resource utilization and performance at the system-level; and at the network-level to address optimal NF placement and routing for service chains, traffic engineering, and load balancing the traffic across *Virtualized Network Function Instance*s (VNFIs); and to provide *High Availability* (HA), *Fault Tolerance* (FT) and *Disaster Recovery* (DR) guarantees.

We begin by presenting *NFVnice*, a userspace NF scheduling framework for *Service Function Chaining* (SFC) to address the system-level resource utilization, performance, and scale challenges. *NFVnice* presents a novel rate-cost proportional scheduling and chain-aware backpressure mechanisms to optimize the resource utilization through judicious *Central Processing Unit* (CPU) allocation to NFs, and improve on the chain-wide performance. It also improves the scalability of NF deployment by allowing to efficiently multiplex multiple NFs on a single core. *NFVnice* achieves judicious resource utilization, consistently fair CPU allocation and provides 2x-400x gain in throughput across NF chains.

Next, in order to address network-level challenges, specifically the orchestration and management of NFs and SFCs we develop *DRENCH* - a novel semi-distributed resource management framework to efficiently instantiate, place and relocate the network functions and to distribute traffic across the active NF instances to optimize both the utilization of network links and NFs. We model *DRENCH* as shadow-price based utilitarian market with *Software Defined Networking* (SDN) controller as a Market orchestrator to solve the *Extended Network Utility Maximization* (ENUM) problem. *DRENCH* results in better load balancing across *Network Function Instance*s (NFIs) and significantly lowers the *Flow Completion Time* (FCT) providing up to 1Ox lower FCT than the state-of-the-art solutions. We also present *Neo-NSH*,

which extends on *Network Service Header* (NSH) to provide a simplified chain-wide steering framework. *Neo-NSH* leverages the SDN controller and discriminates the path-aware chain-wide transport at the control plane and service-aware but instance agnostic routing at the data plane. This separation presents two-fold benefits i) minimize the path management complexity at the SDN controller ii) orders of magnitude reduction in the switch *Ternary Content Addressable Memory* (TCAM) rules; thus it enables for scalable, agile and flexible service function chaining.

Finally, in order to achieve efficient NF migration and to address HA for NF chains, we present *REINFORCE* - an integrated framework to address failure resiliency for individual NF failures and global service chain-wide failures. *REINFORCE* presents a novel NF state replication strategy and distinct mechanisms to provide timely detection of NFs, hardware node (*Virtualized Network Function Manager*), and network link failures; and provides distinct failover mechanisms with strict correctness guarantees. NF state replication exploits the concept of external synchrony and rollback recovery to significantly reduce the amount of state transfer required to maintain consistent chain-wide state updates. Through the optimization techniques like opportunistic batching and multi-phase buffering, *REINFORCE* achieves very low latency (2 orders of magnitude lower latency) and less than 20% performance overheads. *REINFORCE* achieves NF failover within the same node in less than 100μseconds, incurring less than 1% performance overhead; and chain level failover across servers in a *Local Area Network* (LAN) within tens of milliseconds. In addition, we present *REARM*, that adopts the concept of transient VNFs to migrate VNF within and across *Data Center*s (DCs) to facilitate HA in the event of disaster or power outages that frequent the *Green Data Center*s (GDCs).

This dissertation combines abstract mathematical models to describe and derive NFV system behaviors, in order to design and develop system-level implementations for a set of working, ready-to-deploy NFV solutions. Our implementations have demonstrated their superior performance in addressing system-level performance, scale, and failure resiliency challenges. The proposed key solutions have been implemented on OpenNetVM, an open-source NFV framework, and are applicable to other NFV systems due to our generic design.

Acknowledgements

With great pleasure, I would like to acknowledge and wholeheartedly thank all those who have inspired, lead me and been active part of my indelible journey of PhD.

First, I would like to sincerely thank my PhD advisers: Professor Dr. Xiaoming Fu, Professor Dr. K. K. Ramakrishnan, and Dr. Mayutan Arumaithurai, whose support, expertise, continuous guidance, encouragement, and patience has enabled me to author my PhD thesis. I'm a mere mason, for they are the architects who brick by brick have laid the foundation and pillars of my PhD.

Prof. Dr. Xiaoming Fu: I'm extremely grateful for giving me an opportunity to pursue PhD under your guidance. I thank you for all the support, freedom and opportunities you let me to explore and pursue diverse research topics and to visit top research conferences. Your technical guidance and lessons including the art of communication and networking have had an enormous impact on me. I'm immensely grateful for the support and encouragement I've received from you throughout my PhD and also towards seeking my future career in research and academia.

Prof. Dr. K. K. Ramakrishnan: The first person I met before starting my journey of PhD. You are my Guru in every sense and a constant source of inspiration. All my words would simply fail to thank you. Your talk on NetVM during the first summer school was the motivating and shaping part of my research direction. Week after weeks of our hour long discussions have never failed to teach me something new. I thank you for hosting me at Riverside for my valuable secondment.

Dr. Mayutan Arimaithurai: I am lucky to have you as my adviser. Thanks for teaching me the 'Specialization on SDN and NFV' course. You have been more a friend than just my mentor, not just meticulously planning the course of my work, but consistently motivating and guiding at every step of my PhD.

I am also obliged to my thesis defense committee members: Prof. Dr. Klaus Wehrle, Prof. Dr. Dieter Hogrefe, and Prof. Dr. Jans Grabowski. Their comments and suggestions have greatly improved the thesis.

I would also like to thank Prof. Dr. Kai Hwang: my advisor at USC, Prof. Dr. T. H. Sreenivas my advisor at NIE, and Prof. Dr. Panta Murali Prasad, who consistently motivated and encouraged me to pursue PhD.

I would also like to thank Dr. David Koll, who taught me 'Basic SDN' course, the first course of my PhD career which also set the direction for my research. His

teaching, continuous support, and feedback have immensely helped to shape my research work. Also, thanks for making Cleanksy project a memorable one.

I thank all my collaborators, who have helped me during various stages of different projects. Prof. Dr. Timothy Wood, Dr. Sriram Rajagopal, Dr. Jinho Hwang, Dr. Ioannis Psaras, Grace Liu, Wei Zhang, and Argyrious Tasiopoulos: I'm extremely grateful to have worked with you all; all along, I have learnt a lot from each of you. I would also like to thank my labmates at UCR, especially Aditya Dhakal, Ali Mohammadkhan and Mohammad Jahanian, my best companions who made my secondment at UCR a memorable one. I would also like to thank Victor G. Hill for all the timely support on testbed setup at UCR, without whom my work wouldn't have been possible.

I would also like to thank my former and current computer networks group colleagues, especially Dr. Jiachen Chen, Abhinandan S. Prasad, Sripriya S. Adhatarao, Osamah Barakat, Jacopo De Benedetto, and my long term officemate Tao Zhao, whose feedback at different stages has contributed to the quality of this thesis.

I thank Federica Poltronieri, without whom the stay in Germany wouldn't have been pleasant; her immense help in every aspect enabled me to be in Germany as a Roman in Rome, without knowing the ABC of Germany. I am equally thankful and indebted to Annette Kadziora, Gunnar Krull, Tina Bockler, Carmen Scherbaum, and all the staff who have been of great help and support in different matters of need. I would also like to thank our entire Cleansky ITN team, all the advisors, and ESRs who have contributed towards my research progress in one way or another. I would also like to thank the City and the University of Göttingen for providing such a wonderful and serene atmosphere blend with excellent research opportunities.

I would also like to thank all my friends and former colleagues, specially Gadigeppa Malagund, Shailesh Kadamaje, Dr. Manjesh Kumar Hanwal, and Dr. Siddharth S. Bhargav for constant encouragement and fostering the belief that I can!

I thank all my teachers whose lessons have been my guide all along; supremely my parents who have made every effort to ensure that I cake-walkingly step in the right direction in every path of my life. I would also thank my in-laws for their constant encouragement, and bestowing me their crown jewel without whose understanding and support, I wouldn't have taken a bold step to resign my job and join PhD; without whose care, I wouldn't have been able to realize my dream.

I thank the ONE and all who have directly and indirectly helped me and contributed in my march of PhD.

This work was supported in part by the EU FP7 Marie Curie Actions by the EC Seventh Framework Programme (FP7/2007-2013) Grant Agreement No. 607584 (the Cleansky ITN project).

Contents

List of Figures

List of Tables

List of Definitions and Theorems

List of Abbreviations

API	*Application Programming Interface*
BFD	*Bidirectional Forwarding Detection*
BSS	*Business Support Systems*
CapEx	*Capital Expenditure*
CFS	*Completely Fair Scheduler*
COTS	*Commercial-off-the-shelf*
CPU	*Central Processing Unit*
CSFQ	*Core Stateless Fair Queuing*
CSP	*Communication Service Provider*
DC	*Data Center*
DPDK	*Data Plane Development Kit*
DPI	*Deep Packet Inspection*
DR	*Disaster Recovery*
ECN	*Explicit Congestion Notification*
EMS	*Element Management System*
ENUM	*Extended Network Utility Maximization*
ETSI	*European Telecommunication Standardization Institute*
EWMA	*Exponentially Weighted Moving Average*
FCT	*Flow Completion Time*

FIFO	*First-In-First-Out*
FPGA	*Field-Programmable Gate Array*
FT	*Fault Tolerance*
GDC	*Green Data Center*
GPU	*Graphics Processing Unit*
HA	*High Availability*
ICMP	*Internet Control Message Protocol*
ICN	*Information Centric Networking*
ICT	*Information and Communications Technology*
IETF	*Internet Engineering Task Force*
IP	*Internet Protocol*
ISG	*Industry Specification Group*
ISP	*Internet Service Provider*
JIT	*Just-in-time*
LAN	*Local Area Network*
LB	*Load Balancer*
LDP	*Label Distribution Protocol*
LLDP	*Link Layer Distribution Protocol*
LLVM	*Low-Level Virtual Machine*
MANO	*Management and Orchestration*
MPLS	*Multi-Protocol Label Switching*
NAT	*Network Address Translation*
NF	*Network Function*
NFI	*Network Function Instance*

NFV	*Network Function Virtualization*
NFVI	*Network Functions Virtualization Infrastructure*
NFVO	*Network Functions Virtualization Orchestrator*
NS	*Network Service*
NIC	*Network Interface Card*
NPU	*Network Processor Unit*
NSC	*Network Service Chaining*
NSH	*Network Service Header*
NUMA	*Non-uniform Memory Access*
OAM	*Operations, Administration, and Maintenance*
OpEx	*Operational Expenditure*
OS	*Operating System*
OSS	*Operations Support Systems*
PNF	*Physical Network Function*
QoS	*Quality of Service*
RED	*Random Early Drop*
REM	*Random Early Marking*
RFC	*Request for Comments*
RR	*Round Robin*
RTT	*Round Trip Time*
SC	*Service Continuity*
SDC	*Stable Data Center*
SDN	*Software Defined Networking*
SF	*Service Function*

SFC	*Service Function Chaining*
SFF	*Service Function Forwarder*
SFP	*Service Function Path*
SFQ	*Stateless Fair Queuing*
SLA	*Service Level Agreement*
SPDK	*Storage Performance Development Kit*
SPI	*Service Path Identifier*
SR-IOV	*Single Root Input/Output Virtualization*
TCAM	*Ternary Content Addressable Memory*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
URL	*Uniform Resource Locator*
VIM	*Virtualized Infrastructure Manager*
VLAN	*Virtual Local Area Network*
VM	*Virtual Machine*
VNF	*Virtualized Network Function*
VNFI	*Virtualized Network Function Instance*
VNFM	*Virtualized Network Function Manager*
WAN	*Wide Area Network*
DRENCH	*Semi-**D**istributed **Re**source Management Framework for **N**etwork Function **Ch**ains*
REARM	***R**enewable **E**nergy b**A**sed **R**esilient deploy**M**ent of VNFs*
REINFORCE	***RE**sil**I**ent **N**etwork **F**uncti**O**n Se**R**vi**CE** Chains*

Chapter 1

Introduction

Whenever we proceed from the known into the unknown we may hope to understand, but we may have to learn at the same time a new meaning of the word "understanding".

— Physics and Philosophy: The Revolution in Modern Science, 1958.
Werner Heisenberg

Network functionality has significantly evolved beyond the traditional packet forwarding and routing services. Different types of network services have been widely deployed in *Communication Service Provider* (CSP)[1] networks. For example, services to a) enhance network security through *Uniform Resource Locator* (URL) filtering to filter and block malicious web requests, b) improve performance through in-network caching to reduce the load on core network and to reduce access latency from user perspective and c) provide additional value-added services like parental control to block inappropriate web content, and many other services like encryption, compression, *Network Address Translation* (NAT), bandwidth monitors, *etc.*

Traditionally, these network functionalities have been implemented as hardware middleboxes, while the CSPs realize different *Network Service*s (NSs)[2] through the deployment of one or more such middleboxes in their networks. The evolution of the Internet, rapid explosion in the volume and types of services delivered over the Internet/network, the volume of users have contributed and necessitated towards diverse and large-scale deployment of middleboxes. Recent surveys indicate the presence of a diverse set of middleboxes and the volume of middleboxes deployed in CSP networks is on par with the number of switches and routers, constituting about a third of networking devices [1–3]. This diversity and volume of proprietary middleboxes posed several deployment and resource management complexities to the

[1] *Communication Service Provider* includes Telecommunication, Enterprise, *Data Center* (DC), *Internet Service Provider* (ISP) and Cloud that provide & facilitate communication services.
[2] Network service is realized by a well-defined chain of Middleboxes.

network operators [2, 4], *e.g.,* i) high *Capital Expenditure* (CapEx) and *Operational Expenditure* (OpEx) ii) deployment, configuration and management complexities iii) diagnosing performance issues, failures, and recovery from middlebox failures, *etc.*

In order to address and overcome these limitations, in 2012, *European Telecommunication Standardization Institute* (ETSI) proposed the *Network Function Virtualization* (NFV) paradigm to develop and deploy the middleboxes as software based *Network Function*s (NFs) also known as *Virtualized Network Function*s (VNFs) [4].

1.1 Motivation

NFV supplements the benefits of virtualization like reduced hardware costs, faster provisioning, improved availability, disaster recovery, *etc.*, with new opportunities to innovate, deploy and market new network services. Owing to these compelling benefits, ever since the initial inception of NFV, many CSPs, Industry, and Academia have actively pursued and fostered towards the development of NFV. The *Industry Specification Group* (ISG) NFV community has evolved rapidly. At present, the community consists of more than 300 members[3]. This community has contributed from the NFV pre-standardization studies to the detailed specifications and is actively working to develop the required standards for NFV [6].

Alongside, SDN (discussed in §2.1) enables for network programmability through logically centralized intelligence and control allowing the network operators to manage the entire network consistently and holistically, regardless of the underlying network technology [7]. Together NFV and SDN are highly complementary and greatly augment to provide flexible and dynamic softwarized network environment. Most CSPs have already embraced and/or planning to embrace SDN and NFV [8–11].

Although the NFV has gained significant momentum, the recent study and surveys on NFV deployment have pointed out the key problems and challenges hindering the full NFV adoption in CSP networks [12–14]. ETSI ISG NFV and *Internet Engineering Task Force* (IETF) *Service Function Chaining* (SFC) working group have distinctly identified the relevant outstanding problems pertaining to the architecture, management and/or protocol that need to be addressed to enable effective deployment and usage of NFV and realization of SFC in CSP networks [15, 16]. In this work, we seek to study and address some of these critical problems affecting the efficient deployment and realization of network function chains.

[3]Dated: 2018/04/27, Total NFV Members: 127 and NFV Participants 188, includes 38 of the world's major service providers [5].

1.1.1 High Level Research Problems

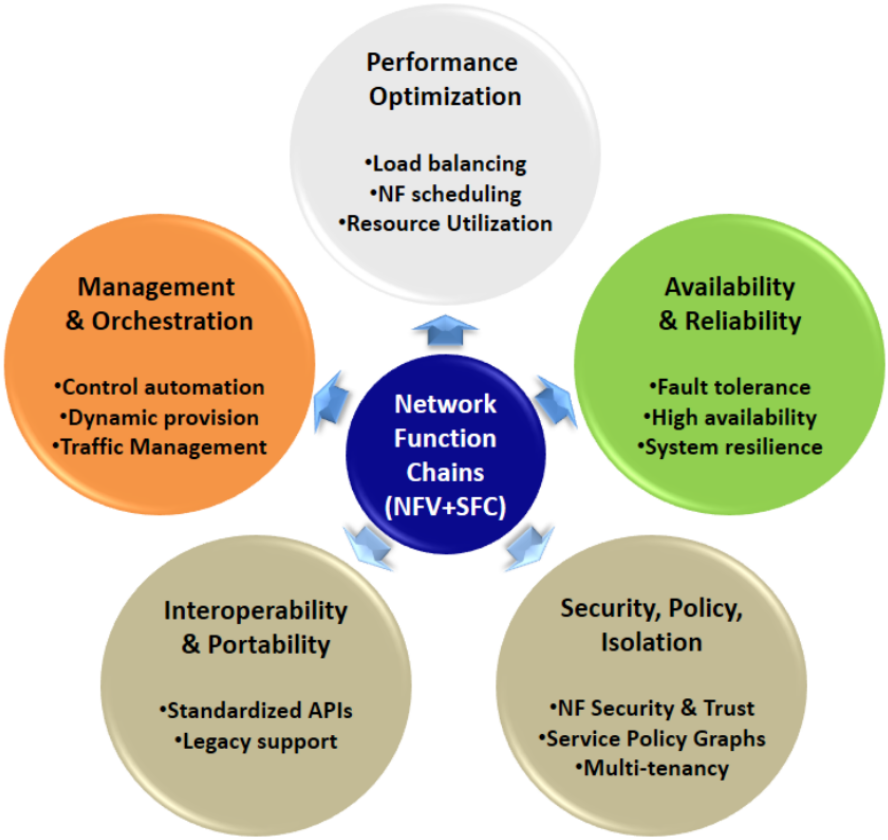


Figure 1.1: High-level Research Problems associated with the Deployment of Network Function Chains.

The deployment and realization of *Network Service Chaining* (NSC) are plagued with several resource management, orchestration, and performance issues. Figure 1.1.1 presents some of the critical high-level research problems associated with NSC, which are briefly discussed below:

P1 Performance Optimization: NFV embraces the use of *Commercial-off-the-shelf* (COTS) hardware *i.e.,* using general purpose computing, storage, and network devices instead of using the dedicated hardware to provide the network services [15]. This greatly benefits to lower the CapEx and also provides flexible deployment options. However, VNFIs based on COTS hardware can encumber severe per-

formance degradation[4] and may not be able to match the throughput, latency, scale, and performance metrics of the dedicated hardware devices that are known to meet the carrier grade performance requirements. Additionally, with SFC where the packets/flows are steered through different network functions in the chain, additional characteristics like memory access and *Non-uniform Memory Access* (NUMA) overheads for processing at distinct NFs within a single physical node, and when the chains span multiple nodes, overhead due to cross node communication, traffic steering, load on VNFIs, network links, also significantly impact the latency and overall chain-wide performance of NSC. Hence, there is a need to account for the aforementioned characteristics for NSC and provide mechanisms to ensure scalability, performance, and efficiency such that the effects on latency, throughput, and processing overhead are minimized.

P2 *Management and Orchestration* (MANO): With NFV, the decoupling of VNFs from the underlying hardware resources engender new management challenges such as end-to-end service to end-to-end NFV network mapping, instantiating VNFs at appropriate locations to realize the intended service, allocating and scaling hardware resources to the VNFs, keeping track of VNF instances location, *etc.* [17]. Management specifically corresponds to *Operations, Administration, and Maintenance* (OAM) of VNFIs and SFC. It includes the mechanism to manage the VNFI, VNFs, network policies, and construction of service paths including the mechanism to perform resource and service monitoring, performance measurement, diagnostic alarm reporting, *etc.* Orchestration corresponds to control automation of the offered NFV services and the underlying resources *i.e.,* to deploy and provision VNFIs instances, to realize SFC, and to control the forwarding behaviors of physical switches using SDN. Control automation is paramount to lower the OpEx and to realize agile NFV. Ensuring correct operation and management of the infrastructure, network functions, and SFCs is vital for the success of NFV. Hence, it is critical to provide consistent management and orchestration framework that can facilitate flexible and dynamic instantiation of VNFIs, placement of VNFs by accounting the infrastructure constraints, traffic and load characteristics of VNFs, service requirements, tenant specific policies and *Service Level Agreement* (SLA) requirements.

P3 Availability and Reliability: NFV platforms are expected to meet the carrier grade availability standards (*i.e.,* greater than or equal to 'five-nines' or 99.999% up time). However, both hardware components in *Network Functions Virtualization Infrastructure* (NFVI) and software VNFs can fail. Additionally, with VNFs the hypervisors can turn out be single-point-of-failurese [18]. Further, with SFC, even

[4]Custom/Proprietary hardware based Network functions typically encompass performance customization and employ acceleration methods, which may not be available in standard hardware.

the failure of any one VNFI in the chain can engender service reliability issues and also result in total service failure. Hence, it is necessary to ensure an appropriate level of resilience to both hardware and software failures. It is also necessary to provide effective mechanisms to provide desired *High Availability* (HA), *Fault Tolerance* (FT) and to tackle service resiliency either via necessary redundancy (hardware and software), replication and consensus mechanisms.

P4 Security, Policy and Trust Management: In NFV, multiple vendors for different NFV elements (*e.g.,* hardware resources, virtualization layer, VNF, virtualized infrastructure manager, *etc.*) may be involved in the delivery and setup of network services [19]. The usage of shared storage, networking, compute devices and interconnectivity among these components add to additional vulnerabilities [20]. Hence, new security and trust issues need to be addressed. In SFC architecture, the static topologically-dependent VNF deployment is replaced with the dynamic chaining of VNF. Hence, the composition of service chain graphs and steering of traffic through these NFs needs to ensure policy compliance and isolation assurances. Additionally, dynamic chaining changes the flow of data through the network, and correspondingly the security and privacy considerations[5] of the protocol and deployment will need to be reevaluated [21]. Hence, to tackle the increasing security threats NFV platform needs to provide a comprehensive and effective approach to secure the NFVI, *Virtualized Infrastructure Manager*s (VIMs) to build secure execution platform for the NFs.

P5 Interoperability and Portability: Interoperability of the new VNFs with the existing dedicated and proprietary hardware based network functions or *Physical Network Function*s (PNFs) is necessary to ensure legacy device support. Also, the ability run the virtual appliances from different vendors in different but standardized DC environments of different operators is necessary. Hence, the key challenge is to define a unified interface to decouple the VNFIs from the underlying hardware and to promote distinct yet interoperable ecosystem for both VNF vendors and DC vendors [4].

Hence, it is necessary to re-consider and address these problems to realize successful deployment of *Network Service Chaining* and reap the benefits of network softwarization with SDN and NFV.

[5] As user traffic (network flows) is subject to processing at multiple VNFIs from different vendors, it is necessary to ensure right access control privileges to avoid the breach of trust between the users, service providers, and VNF vendors.

1.2 Research Goals

In this dissertation, we intend to discern and address few of the NSC problems outlined in section §1.1.1. We particularly seek to develop the NFV resource management framework and distinct mechanisms towards resolving the following *Service Function Chaining* problems in SDN and NFV based networks:

G1 Performance Optimization: We specifically seek to account the aspects of VNF resource allocation, especially the *Central Processing Unit* (CPU) resource for efficient multiplexing and scheduling of NFs to address scalability and to improve performance through efficient NF scheduling, judicious and fair chain-wide resource allocation by accounting the *Network Service Chaining* characteristics.

G2 Management and Orchestration: We seek to build a low complexity resource management and orchestration framework to address dynamic provisioning, placement and lifecycle management of NFs. We also seek efficient and scalable solutions to address *Network Service Chaining* and traffic management *i.e.,* steering the traffic through a chain of network functions by accounting the congestion in the network and load on the *Network Function Instance*s (NFIs) involved in the service chain.

G3 Availability and Reliability: We seek to address the two distinct aspects of service continuity i) to provide fault-tolerance and service resiliency in the case of VNFI resource failures and ii) to address *Disaster Recovery* (DR) and to provide high availability in the case of power outage within or across DCs. In this, we distinctively seek efficient mechanisms for achieving NF Resiliency via redundancy, fault-tolerance, and NF migration.

Overall, to address the above specified distinct goals, we seek to build a resource management framework in line with the ETSI NFV-MANO reference architecture (illustrated in Section §2.2.1).

Dissertation Statement: Our primary objective is to devise solutions towards realizing an efficient, scalable and reliable framework for NF chains.

Towards this objective, we seek to develop NFV-MANO framework, especially the *Virtualized Network Function Manager* (VNFM), *Network Functions Virtualization Orchestrator* (NFVO) components and the *Element Management System* (EMS) for the *Virtualized Network Function Instance*s (VNFIs) to improve scalability, performance, resource-utilization efficiency, and resiliency of deploying the NF chains in SDN/NFV ecosystem.

1.3 Research Challenges

This section outlines the key challenges in addressing the NSC problems (§1.1.1), and realizing our research goals described in Section §1.2.

1.3.1 Performance Optimization

High-performance NFV platforms employ kernel bypass techniques like *Data Plane Development Kit* (DPDK), *Single Root Input/Output Virtualization* (SR-IOV), Netmap to achieve and meet line rate packet processing. In order to achieve high throughput and low latency, they avoid the interrupt overheads and perform poll mode operation on *Network Interface Card*s (NICs), which requires a dedicated core for each NF. This approach not only limits the scalability, *i.e.,* the number of NFs that can be run on a server, but also result in wastage and inefficient utilization of resources, especially when the workload is low.

To improve on resource utilization, approaches such as NF consolidation and multiplexing of NFs on a single server node have been proposed [3,22]. However, the NFs exhibit diverse processing (both computation and I/O) characteristics. Hence, the key challenge, especially with the multiplexed approach is to ensure fair allocation of CPU resources by accounting both CPU and I/O requirements of the NFs.

Also, with SFC, where the NFs running on different dedicated cores process packets in a specific order can encumber expensive cross-core communication and cache access overheads resulting in severe degradation of chain-wide performance. Hence, to ensure chain-wide performance, the NUMA characteristics of the node need to be accounted. In addition, processing a packet at one or more NFs in the chain, only to have it dropped from a subsequent bottleneck's queue is wasteful. Hence, beyond simply allocating CPU time fairly to NFs, an additional challenge is to account for the impact of bottlenecks across SFC in allocating the CPU resources.

1.3.2 Management and Orchestration

As networks grow in scale and complexity, traffic dynamics change and trigger for reallocation and reconfiguration of network resources. In case of high demands, some resources end up being over-utilized, resulting in higher latency and SLA degradation, while on other occasions, end up being underutilized. Further, in such circumstances, in order to meet the performance and energy objectives, the NF instances need to be *dynamically instantiated, decommissioned or even relocated/migrated.* This necessitates the need to manage and orchestrate a large number of diverse NFs

by accounting for both network resource (topology) and traffic characteristics.

In addition, the *Service Function Chaining* characteristics for desired service policies on arbitrary network topologies also need to be accounted to correctly manage and orchestrate the VNFIs. This must also ensure to avoid unnecessary routing of traffic within the network which can result in over utilization of network links at the cost of service degradation.

Additionally, the VNFI management and orchestration in the presence of *Service Function Chaining*s for arbitrary network topologies need to be addressed. However, such traffic aware NF placement, balancing the load in network and across NFs are known to be NP-hard problems [23, 24]. Also, traffic dynamics, especially in the DCs can change at very fine-grain timescales (order of seconds) [25]. Hence, the core challenge is to ensure an adaptive and incrementally deployable solution that is both sufficiently optimal and swift.

1.3.3 High Availability and Failure Resilience

Hardware resources (including the network links and servers hosting the network functions), and software network functions are prone to failures. Any such failure, can partially or completely disrupt the network services. To avoid service outages it is necessary to incorporate *High Availability* (HA) and service failure resiliency mechanisms. However, providing HA support for softwarized NFs and chain of NFs can result in significant resource overheads and performance penalty on normal operation. Hence, the main challenge in providing HA and Failure resiliency is to ensure the performance during normal operations is not adversely impacted and also to mitigate the network resource overheads.

Additionally, most of the NFs are stateful entities that actively maintain, update and rely on the current state to process the packets. Hence, to maintain service correctness and to enforce correct packet processing, consistent NF state needs to be preserved across the instances. This requires careful mechanisms to effect consistent NF state migration.

Also, different kinds of failures posit different resiliency characteristics. For example, it is easier to detect and react to fatal software failures (system crashes) than to detect and address functionality based errors, likewise fault containment and isolation for different hardware failures differ. Further, it is necessary to account both individual NF failures (software failure) and NF chain level failures due to hardware (link and node) failures, so that the solution can optimally detect and react to distinct failures.

Alongside, with the increasing electricity demands in the *Information and Communications Technology* (ICT) sector, the inclination towards employing renewable (green) resources to power up the data centers is also increasing [26, 27]. However, the Green energy supply is rather intermittent and unstable, which can result in power outages resulting in service disruptions [28, 29]. Hence, ensuring HA and providing FT of VNFs in the event of such disasters is also necessary.

1.4 Summary of Contributions

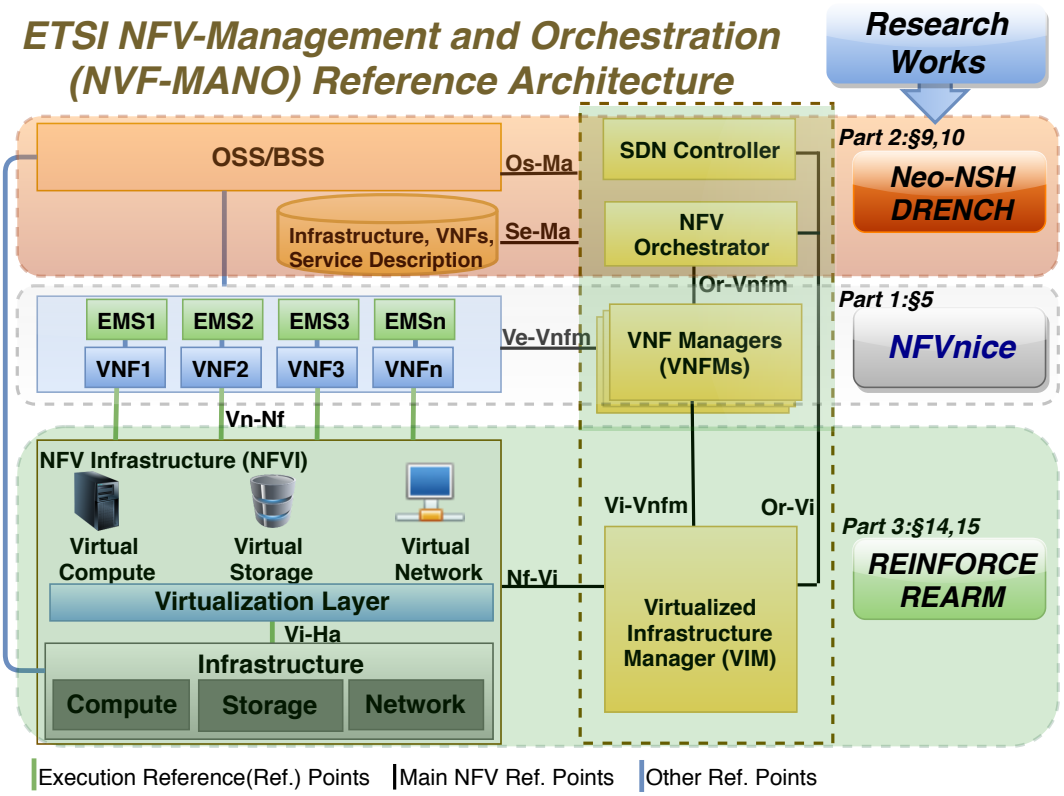


Figure 1.2: Research Contribution in the realm of ESTI NFV-MANO Reference Architecture for Network Service Chains.

This dissertation presents a resource management framework to realize efficient, scalable and reliable *Network Service Chaining*. Our framework is based on the ETSI MANO reference architecture [17] and is aimed towards realizing the goals outlined in §1.2. Figure 1.2 outlines and maps the contributions of this dissertation in the context of the ETSI NFV-MANO reference architecture.

1.4.1 Performance Optimization

We developed **NFVnice** [30] to account for the system level performance and scale challenges outlined in Section §1.3.1. NFVnice is a user space NF scheduling and service chain management framework that provides fair, efficient and dynamic resource scheduling capabilities on NFV platforms. NFVnice enables to multiplex running multiple NFs on a single core, thus it improves the resource utilization and scalability of NF deployment on a server node. Further, it allows to deploy chain of NFs on a single core and schedule them appropriately by accounting the NUMA overhead, which results in judicious resource utilization, avoidance of wasted work across NF chain and significantly improves the NF chain performance. NFVnice is capable of controlling when network functions should be scheduled and improves NF performance by complementing the capabilities of the OS scheduler but without requiring changes to the OS's scheduling mechanisms. NFVnice leverages *cgroups* - a user space process scheduling abstraction exposed by the Linux operating system and provides the appropriate rate-cost proportional fair share of CPU to NFs. NFVnice monitors the load on a service chain at high frequency (1000Hz) and employs backpressure to shed load early in the service chain, thereby preventing wasted work. Through rate-cost proportional scheduling, CPU shares of the NFs are computed by accounting the heterogeneous packet processing costs of NFs, I/O, and traffic arrival characteristics.

Our controlled experiments demonstrate that when compared to default Operating System schedulers, NFVnice is able to achieve judicious resource utilization, consistent fairness and 2x-400x gain in throughput across NF chains. NFVnice achieves this even for heterogeneous NFs of varying chain lengths, with vastly different computational costs and for heterogeneous workloads.

1.4.2 Management and Orchestration

We developed **DRENCH** [31] to address the network-wide orchestration and management challenges outlined in Section §1.3.2. In DRENCH, we consider an NFV market with a centralized SDN controller that acts as the market orchestrator of NFV nodes, and through competition, the NFV nodes effect flow steering, service instantiation, and consolidation decisions. DRENCH orchestrator parameterization strikes the right balance between optimizing the path stretch and balancing the number of active VNFs and load across these active instances. DRENCH results in better load balancing across NFIs and significantly lowers the *Flow Completion Time* (FCT), providing up to 1Ox lower FCT than the state-of-the art solutions.

To address efficient and scalable routing construct with *Network Service Header*

(NSH), we present *Neo-NSH* [32] to provide a simplified chain-wide steering framework by extending on the NSH [33] - a recent IETF *Request for Comments* (RFC)[6] for realizing the network service plane. *Neo-NSH* leverages the SDN controller and discriminates the path-aware chain-wide transport at the control plane and service-aware but instance agnostic routing at the data-plane. This separation presents two fold benefits i) minimize the path management complexity at the SDN controller ii) orders of magnitude reduction in the switch *Ternary Content Addressable Memory* (TCAM) rules; thus it enables for agile and flexible service function chaining.

1.4.3 High Availability and Failure Resilience

To address NF resiliency challenges outlined in Section §1.3.3 and to account both individual NF failures (software failure) and NF chain level failures due to hardware (link and node) failures, we present an integrated high-availability framework for DPDK based containerized NFs. In **REARM** [34], we specifically study the impact of deploying VNFs in *Green Data Center*s (GDCs) and make a case for addressing the VNF reliability and high availability to effectively tackle the stability concerns of GDC. REARM outlines a simple NF Migration framework that accounts the NF service chain characteristics and adapts the NF state migration to reduce both computation and communication overheads for maintaining the remote NF replicas. REARM adopts the concept of Transient VNFs that rely on a very short advance warning time to seamlessly migrate the VNFs from GDC to a more reliable and stable *Data Center*[7].

In **REINFORCE** [35], we implement a full-fledged framework incorporating the NF manger, NFs, and common NF services library *libnf* to quickly detect and react to different failures and develop distinct failover mechanisms that identify and prioritize the migration of specific NF states such that the overall operational framework incurs minimal performance overhead and ensures state correctness guaranty across NF chains. Compared to the state-of-the-art solutions, REINFORCE achieves significant reduction (2-3 orders of magnitude) in recovery time, latency impact during normal operation and maintains 85-90% of the normal operation throughput.

Summary These distinct components enable to resolve both system and network-wide performance, scale and reliability concerns in the deployment and management of NF chains.

[6] NSH became RFC '`RFC8300`' on 12-Jan-2018; at the time of our proposal, it was an IETF draft version 04.

[7] DCs powered by non-renewable (brown) energy; also known as *Stable Data Center*s (SDCs).

1.5 Dissertation Outline

This section outlines the three parts of this dissertation and the organization of chapters within these parts. In Chapter §2, we first present the background on state-of-the-art SDN/NFV frameworks, and briefly introduce the ETSI NFV-MANO framework, high performance NFV platforms and NF deployment options in realizing the NF chains and outline the key system level and network-wide challenges n deployment of NSC.

In Part I, we present the *Virtualized Network Function Manager* (VNFM) and *Network Function* (NF) level management framework to address and overcome the system level challenges like scalability, performance, resource-utilization, isolation, and fairness. Chapter §3 outlines the problem statement, Chapter §4 presents the state-of-the-art solutions and related work and Chapter §5 details our solution NFVnice, which serves as a tunable user-space scheduling framework for NFs.

In Part II we present the resource management framework to account for the network-level challenges associated with SDN and NFVI and facilitate towards dynamic network function placement and VNFI instantiation. Chapter §7 outlines the problem statement, Chapter §8 presents the state-of-the-art solutions and related work and Chapter §9 details our resource management framework to account NF placement, life-cycle management, and load balancing, and the Chapter §10 presents the routing scheme to facilitate SFC.

In Part III we present the resiliency and NF state migration framework. Chapter §12 outlines the problem statement, Chapter §13 presents the state-of-the-art solutions and related work. In Chapter §14 we present the details of NF and NSC failure resiliency and NF state migration mechanism and in Chapter §15, we present our solution to tackle the reliability issues arising due to intermittent renewable energy powered DCs.

And finally, in Chapter §17, we revisit the overall contributions and impact of this dissertation and outline the key future research prospects of this dissertation. In addition, the supplementary materials in support of this dissertation including the relevant pseudo code, proof of theorems, data-flow and sequence diagrams are listed in the appendix Chapters §*A-D* of part IV.

Chapter 2

Background

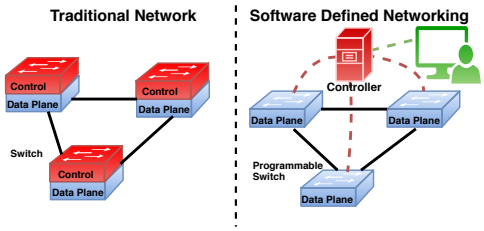
In this chapter, we provide the fundamental concepts that serve as necessary prerequisites for comprehending the subsequent parts and chapters of this dissertation. First, we briefly present the SDN, NFV and SFC concepts, and introduce NSH.

We then present the ETSI NFV-MANO architecture which serves as the basic template for all our research components and also briefly discuss the reference NFV platforms. We present the background on high performance packet processing engines, NFV platforms and scheduling framework that serve as prerequisites to subsequent Chapters.

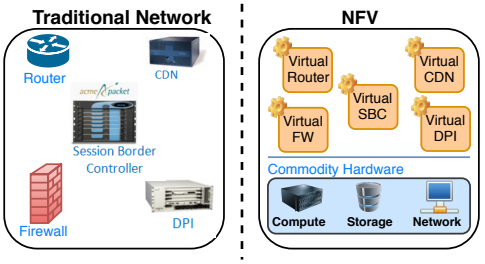
Contents

2.1 Network Softwarization: SDN and NFV

The advent of "***Network Softwarization***" primarily in the form of *Software-Defined Networking* (***SDN***) and *Network Function Virtualization* (***NFV***) has shaped and accelerated the transformation of networking landscape and fostered incessant innovation in design, deployment, and management of networking infrastructure. Network Softwarization is expected to revolutionize the way network and computing infrastructures are designed and operated to deliver services and applications in an agile and cost effective way [36].



(a) SDN Architecture (Transition from ossified network control to Programmable, logically centralized control plane).



(b) NFV Architecture (Transition from dedicated hardware/software to softwarized services with commodity hardware).

Figure 2.1: Illustration of Software-Defined Networking and Network Function Virtualiztion Architecture.

2.1.1 SDN

SDN architecture (shown in Figure 2.1a) decouples the network control plane from the forwarding data plane and provides a logically centralized controller which enables to remotely control and configure the forwarding behavior (flow tables) on different networking devices (switches and routers). This separation and control over the packet forwarding behavior from a logically centralized controller vests network administrators with the flexibility to enforce network-wide policies and to perform dynamic orchestration of network traffic (flows) and networking resources. Thus SDN caters towards agile, programmable and flexible networking architecture.

2.1.2 NFV

In 2012, ETSI proposed the NFV paradigm which extends the standard virtualization to the networking infrastructure (shown in Figure 2.1b). NFV decouples the

software implementation of NFs from dedicated proprietary hardware (compute, storage and network) resources and enables to run the NFs on commodity off-the-shelf server machines. Thus, beyond the compelling benefits of virtualization like reduced hardware costs, faster provisioning, improved availability, disaster recovery, *etc.*, NFV also provides the opportunity to innovate, deploy and market new network services utilizing one or more network functions.

SDN and NFV are highly complementary and greatly augment to provide flexible and dynamic softwarized network environment to realize service function chains [37]. In the subsequent section, we will present the fundamental NFV-SDN reference architecture.

2.2 NFV Reference Architecture and Platforms

In this section, we present the ETSI's NFV-MANO Reference Architecture and also the corresponding reference open-source implementations and NFV platforms.

2.2.1 ETSI NFV-MANO Architecture

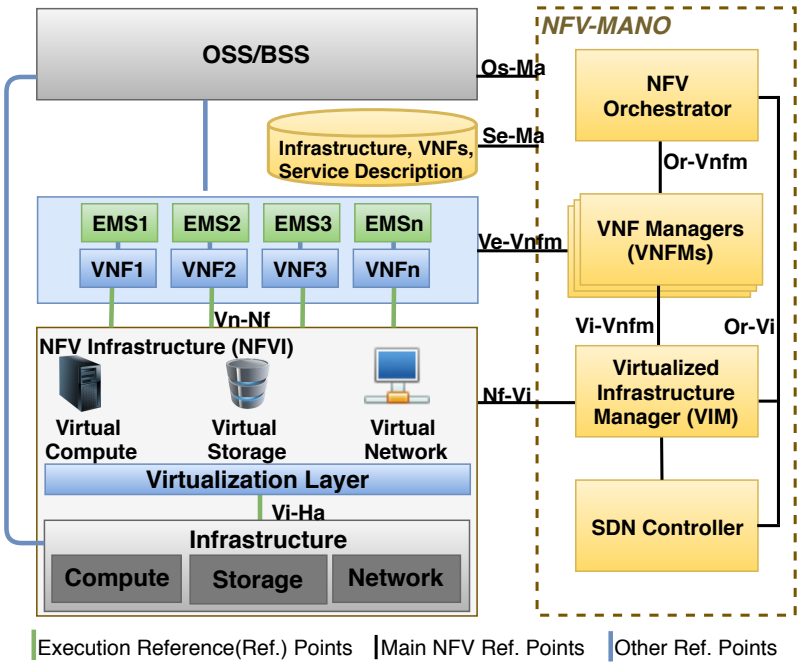


Figure 2.2: ETSI's NFV-MANO Reference Architecture

ETSI proposed the reference architecture for NFV *Management and Orchestration* [17] as shown in Figure 2.2. We briefly describe the key components and their associated roles.

- *Network Functions Virtualization Orchestrator* (NFVO) is responsible for the creation and management of end-to-end services. NFVO functionality can be divided into two broad categories:
  i) Resource orchestration: to provide services that support accessing NFVI resources in an abstracted manner independent of any VIMs, as well as governance of VNFIs that share the NFVI resources.
  ii) Service orchestration: to create end-to-end services by composing different VNFs, including the topology management of the network service instances.

- *Virtualized Network Function Manager* (VNFM) is responsible for the life-cycle management of one or more VNFIs. There can be more than one VNFMs in a single domain and single VNFM can be associated with one or more VNFIs of same or different types.

- *Virtualized Infrastructure Manager* (VIM) is responsible to manage and control the NFVI, *i.e.,* all the physical and virtual resources in a single domain. Multi-site NFV architecture may encompass more than one VIMss, with each of them managing or controlling part of the NFVI resources.

- SDN controller: is responsible for the orchestration and configuration of network forwarding rules on the networking elements of NFVI. The exact placement and interaction of the SDN controller with other components of MANO can vary [38] based on the deployment factors.

In addition, it proposes three distinct kinds of reference points[8] to facilitate interaction amongst these components depending on the type of functional connectivity and interaction perceived in the MANO.

2.2.2 Reference Implementations and NFV Platforms

There have been several closed-group and open-source initiatives taken up towards realizing the ETSI NFV-MANO. We'll briefly discuss on a few of the most related reference implementations and platforms.

1. ***OPNFV***[9]: Open Platform for NFV (OPNFV) is an open source collaborative

[8] Refers to a conceptual point at the conjunction of two communicating functional entities.
[9] https://www.opnfv.org/

project founded and hosted by the Linux Foundation. OPNFV integrates components from several open source upstream projects, such as OpenStack, OpenDaylight, FD.io, and many others [39]. It has evolved from supporting traditional *Virtual Machine* (VM) based NFV deployment to facilitating containerized NFs.

The highlight of OPNFV is the Doctor project[10] that aims to build fault management framework for high availability of Network Services on top of the virtualized infrastructure. The current fault management release relies on OpenStack Ceilometer components to detect different faults and initiate failover. It can be noted that the current order of recovery is in tens of seconds, and are actively working towards improving the overall failover time.

2. ***CloudNFV***[11] is an open platform for implementing NFV based on cloud computing and SDN technologies in Cloud (multi-vendor environment). OpenStack framework and components are leveraged for NFV orchestration. CloudNFV in start contrast to ETSI NFV-MANO considers a unified data model for both management and orchestration [40].

3. ***OpenMANO***[12]: OpenMANO is an open source project led by Telefónica. It consists of `openmano, openvim` components and additionally consists of a graphical user interface component `openmano-gui` to interact with `openvim` and `openmano` components. It targets traditional VM based NFIs on standard Linux servers, OpenFlow switches, and SDN controller. In addition, `openvim` *Application Programming Interface*s (APIs) also support for OpenStack based VIM and facilitate services including the creation and deletion of VNF templates, VNFIs, network service templates and network service instances to realize SFC.

2.3 Service Function Chaining

In order to fulfill different network/operator policies and to enrich user experience through different in-network service capabilities, various NFs are often chained together. SFC defines an ordered set of abstract service functions and ordering constraints that must be applied to specific packets and/or flows [21]. Figure 2.3 depicts typical network policies where different network services (*e.g.,* Firewall, Load balancer, Video optimization, *etc.*) are applied in a specific sequence.

In the subsequent section, we'll present the recent IETF RFC*8300* that enables to realize a dedicated service plane for SFC. This section serves as the prerequisite

[10] https://wiki.opnfv.org/display/doctor/Doctor+Home
[11] http://www.cloudnfv.com/
[12] https://github.com/nfvlabs/openmano

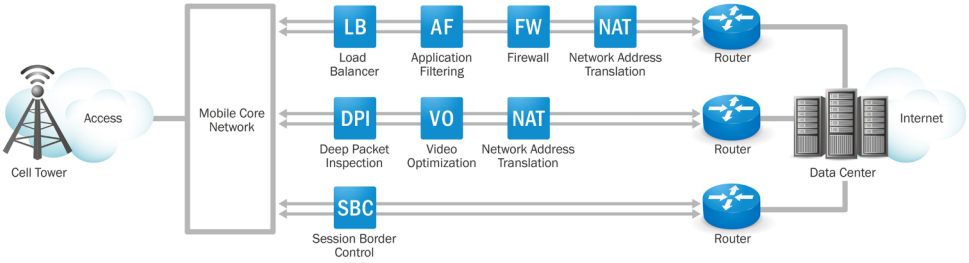


Figure 2.3: Example use case of Service Function chaining in Telecommunication.

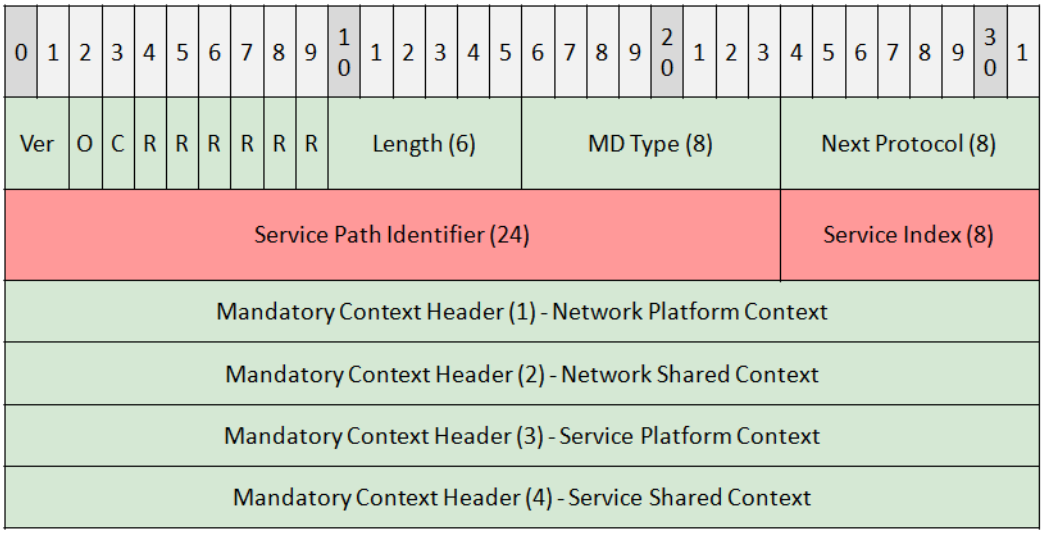
for our work in Chapter §10.

2.3.1 NSH - Dedicated Service Plane for SFC

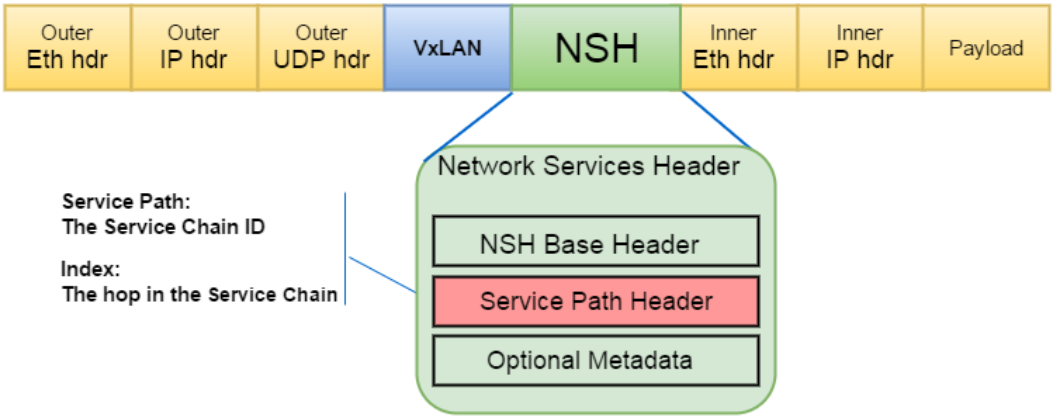
Network Service Header (NSH) [33] is an IETF RFC[6] to address *Service Function Chaining* (SFC). As outlined in the RFC7665 [41], NSH facilitates routing across *Service Functions* (SFs) based on the SFC specific packet encapsulation. NSH defines the data plane header format to create a dedicated service plane for realizing SFC. The NSH as shown in Figure 2.4a consists of the following fields:

- A 4-byte Base header consisting of Version, Flags, MD Type and Next Protocol fields. The next protocol field indicates the protocol type of the encapsulated data.
- A 4-byte Service Path Header consisting of a 24 bit *Service Path Identifier* (SPI) and 8 bit *Service Index* (SI), is used to define the service path that interconnects needed service functions.
- Metadata context headers. The value of MD Type determines the context headers. If the value is 0x1, it consists of four mandatory 32-bit context headers as shown in Figure 2.4a or if the value is 0x2, this field is optional, consisting of variable length context headers.

SPI defines one of the possible instantiations *i.e.,* a logical path to the sequence of specific service function instances of an SFC, while the SI indicates the location within the service path. In addition, NSH defines optional header fields that can carry metadata information. The format of metadata is determined by the MD type field in the base header. However, it must be noted that NSH needs to be inserted onto encapsulated packets, *i.e.,* the actual transport/steering of packets in the network is based on the outer encapsulation.



(a) Packet Structure of Network Service Header.



(b) NSH usage with VXLAN Encapsulation

Figure 2.4: Packet Structure of Network Service Header and Usage with VXLAN Encapsulation.

2.3.2 Benefits of NSH

The NSH approach of specifying a dedicated service plane for service function chaining offers several benefits:

- NSH provides a transport and topology independent service forwarding framework. This decoupling enables the service plane to be realized as overlay service over the existing data plane without requiring any additional complexity and protocols at the data plane.
- NSH enables the ability to classify and re-classify the flows at each service functions. This enables to dynamically steer same flows across different service paths and enables to have richer and finer policy control.
- NSH enables to exchange metadata across service functions in a chain through the context header fields. This aspect is beneficial for Gi-Lan/mobile use cases that can carry the subscriber ID and Tenant IDs across the chain to realize per-user/per-subscriber based policies.
- NSH also provides end-to-end service path visibility. This enables to monitor and troubleshoot service functions, which is critical for OAM to support high availability and resiliency.

2.3.3 How does NSH work?

NSH relies on outer transport encapsulation such as *Multi-Protocol Label Switching* (MPLS), VLAN/VXLAN, GRE to transport the packets. Figure 2.4b shows the usage of VXLAN as a transport encapsulation for NSH. The VXLAN and outer headers must be set accordingly to indicate the presence of NSH.

Upon classification of packets (typically the role of SDN controllers or the dedicated service classifiers), NSH must be inserted to the original packet along with the outer transport encapsulation. The Service path header of NSH determines the list and order of execution of the service functions.

*Service Function Forwarder*s (SFFs) *i.e.,* NSH aware network switches, then examine the NSH header, specifically the service path header and forward the packets towards the next intended function using the combination of SPI and *Service Index (SI)*. After executing the intended service, SI is decremented either by the NSH aware SFs or the NSH aware proxies. Once the last service function is executed, the NSH can be removed from the packet and forwarded towards the intended destination. Thus NSH creates a dedicated service plane which is independent of the

underlying transport and facilitates forwarding across service function instances of the chain.

2.4 High Performance Packet Processing

In this section, we briefly present the key developments in the field of high performance packet processing with the COTS-hardware. This section is essential and serves as a basic introduction to the NFV platform implemented in Chapter §5, §14 and §15.

Softwarized NFs, SDN, and hypervisor based switching technologies have been stymied by the performance achievable with commodity servers [3]. These limitations on throughput and latency have prevented the VNFs from supplanting custom designed hardware. The major bottlenecks being:

- Interrupt driven processing: Interrupts are generally used to notify an *Operating System* (OS) about the packet is ready for processing. However, interrupt handling is expensive in modern superscalar processors [42]. When the packet reception rate increases, this results in increased interrupt activity, which further expend the CPU cycles resulting in very low throughput in such systems.
- Buffer Copy Overhead: The OSs read incoming packets into kernel space and then copy the data to user space (to the socket buffer of the associated application). These extra copies of packet buffers incur greater overhead especially in the virtualized environments (where the copy from hypervisor/host OS to guest OS followed by copy to application buffer) [43].
- Complex networking stack: The networking stack in Linux has evolved significantly over the years. Any packet processing in Linux network stack incurs several functions incurring expensive CPU cycles. It has been shown that networking stack accounts to large share (roughly 75%) of CPU cycles necessary to put the packet on the wire [44].

2.4.1 Kernel Bypass Approaches

Several alternatives have been proposed to achieve high performance networking with COTS hardware that tend to overcome the aforementioned concerns. We briefly outline a few of the associated packet I/O technologies and frameworks.

- *Data Plane Development Kit* (DPDK) is a networking framework written in *C*. It provides a set of libraries and drivers that provide a kernel-bypass technique for fast packet processing in user-space on native Linux systems [45]. It provides

poll-mode NIC drivers that enable the user space application to efficiently process the packets by bypassing all the overheads: kernel interrupts, copy overheads and networking stack processing[13].

- *Netmap* is also a rich hardware independent, networking framework implemented as a Linux kernel module. It allows the userspace applications a very fast channel to exchange raw packets directly with the network adapter bypassing the kernel overheads due to buffer copy, interrupt processing and networking stack processing [44].

- *PF_RING*[14] is a high speed packet processing technique that makes use of shared (memory mapped) circular ring buffers between the user space application and the kernel drivers. It enables to bypass the kernel networking stack, and avoid the buffer copy to provide packets directly to the user space applications. This feature is made available from Linux kernels 2.6.32 onwards.

Table 2.1: State-of-the-art High Performance NFV Platforms

NFV Platforms	Packet I/O Framework	NF Runtime	Remarks
OpenNetVM [46]	DPDK pipelined execution	Docker Container or Linux process based NFs	Poll Mode NFs implemented in C supports NF Chaining
ClickOS [47]	Netmap pipelined execution	Unikernel: MiniOS and Click elements	Single threaded NFs implemented in C/C++ supports NF Chaining
NetBricks [48]	DPDK run-to-completion	Rust based NFs with LLVM runtime	Poll Mode NFs implemented in Rust single process NF Chain
HyperNF [49]	ptnetmap and virtual i/o pipelined execution	VM based NFs with XEN hypervisor	Multi threaded VMs implemented in C supports NF Chaining

2.4.2 Reference Implementations and Platforms

Over the past few years, several high performance NFV platforms have been realized by making use of these kernel bypass I/O frameworks. Table 2.1 lists some of the state-of-the-art high performance NFV platforms.

We consider building our NFV resource management framework by leveraging

[13]DPDK is one of the best open source projects with rich and elaborate documentation that I have come across. For any additional information readers are encouraged to visit the site: `https://dpdk.org/doc`

[14]https://github.com/ntop/PF_RING

these high performance NFV platforms. We chose OpenNetVM [46] as the framework of choice for realizing the resource management framework.

2.5 Scheduling in Linux

In this section, we present a brief overview on process scheduling in Linux *Operating System* (OS). The Linux scheduler is modular and provides different scheduling classes and different process scheduling policies. Table 2.2 lists different scheduling policies and the priority configurations that determine the scheduling characteristics for different scheduling policies.

Table 2.2: Linux Scheduling Class and Policies in kernel v4.4.0

Scheduling Class	Scheduling Policy	Priority Range		Remark
		Static	Dynamic	
CFS	Default	0	[-19, 19]	fine grain fair sharing
	Batch	0	[-19, 19]	coarse grain fair sharing
Real Time	RR	[1, 99]	-	time sliced 1-100 milli seconds
	FIFO	[1, 99]	-	no time slicing
Idle	Idle	0	least	Least priority and Idle tasks

Scheduling class is an extensible hierarchy of scheduler modules that encapsulate the scheduling policy details. They offer an interface to the main scheduler skeleton implemented through the `sched_class` structure that defines the behavior of the scheduler. This provides the flexibility to easily extend and implement multiple scheduling policies within the same scheduling class.
Scheduling policy determines the actual scheduling behavior. Within the same scheduling class, there can be different policies that eventually determine the scheduling characteristics of the processes.
Priority: In Linux, every process is assigned a priority value, and the scheduling decisions are based on the process priorities. Priorities are classified into dynamic (can be assigned by the user during the runtime or updated by the kernel during the runtime of the process) and static (assigned by the user at the time of creation of the process, but can be changed during runtime).

Completely Fair Scheduler *(CFS)*: The CFS class of schedulers use a nanosecond resolution timer to provide fine granularity scheduling decisions. Each task in CFS maintains a monotonically increasing virtual run-time which determines the order and quantum of CPU assignment to these tasks. The time-slice is not fixed but is determined relative to the run-time of the contending tasks in a time-ordered

red-black tree [50,51]. CFS tuning parameters are presented in Appendix §B.2. The task with the smallest run-time (the left most node in the ordered red-black tree) is scheduled to run until either the task voluntarily yields or consumes the allotted time-slice. If it consumes the allocated time-slice, it is re-inserted into the red-black tree based on its cumulative run-time consumed so far. The CFS scheduler is analogous to weighted fair queuing (WFQ) scheduling [52,53]. Thus, CFS ensures a fair proportion of CPU allocation to all the tasks. CFS Default is also known as CFS normal scheduler. The CFS Batch variant has fewer timer interrupts than normal CFS, leading to a longer time quantum (1ms) and fewer context switches. Since the release of kernel 2.6.23., the CFS [54] is the default scheduler.

Real Time Scheduling Linux supports soft real-time scheduling and provides two scheduling policies namely i) *Round Robin* (RR) and ii)*First-In-First-Out* (FIFO). The processes are assigned static priorities in the range 1 (low) to 99 (maximum). As real-time tasks aim at deterministic runs, they are always given preference over the CFS and Idle class of schedulers. While FIFO scheduler has no fixed time slice, RR simply cycles through processes with a specified time quantum[15], but does not attempt to offer any concept of fairness.

2.5.1 Control Groups

Cgroups Also known as control groups is a Linux kernel feature that allows to account, limit and isolate the resource usage *i.e.,* CPU, memory disk I/O, network, *etc.* of a collection of processes [55]. This mechanism enables a collection of processes to be bound by a set of limits or parameters defined via the cgroup filesystem. Cgroups along with namespaces are the fundamental building blocks of linux containers [56].

Cgroups provide the minimum essential kernel mechanisms required to efficiently implement process aggregations to provide group specific resource sharing and isolation. The grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (memory, CPU, network, disk I/O) [55].

Linux provides two variants namely Cgroup-v1 [56] and Cgroup-v2 [56]. In our work, we make use of Cgroup-v1's CPU subsystem to facilitate weighted fair share of CPU for the contending NFs. Details of Cgroup setup are presented in Appendix §B.1.

[15]The default time slice of 100 msec on 14.04 low-latency profile. It can be configured via `sched_rr_timeslice_ms` parameter

Part I

Addressing System-level Challenges in NFV Resource Management: Performance and Scale for Network Functions

Chapter 3

Problem Statement

In this Chapter, we present the performance, scale, and resource utilization efficiency problems that exist with *Network Service Chaining* (NSC) and outline the key system level challenges that need to be addressed to overcome these problems.

Contents

3.1 Introduction

Network Function Virtualization (NFV) seeks to implement network functions and middlebox services such as firewalls, NAT, proxies, deep packet inspection, WAN optimization, etc., in software instead of purpose-built hardware appliances. These software based network functions can be run on top of commercial-off-the-shelf (COTS) hardware, with virtualized network functions (NFs). Network functions, however, often are chained together [21], where a packet is processed by a sequence of NFs before being forwarded to the destination.

The advent of container technologies like Docker [57] enables network operators to densely pack a single NFV appliance (VM/bare metal) with large numbers of

network functions at runtime. Even though NFV platforms are typically capable of processing packets at line rate, without efficient management of system resources in such densely packed environments, service chains can result in serious performance degradation because bottleneck NFs may drop packets that have already been processed by upstream NFs, resulting in wasted work in the service chain.

NF processing has to address a combination of requirements. Just as hardware switches and routers provide rate-proportional scheduling for packet flows, an NFV platform has to provide a fair processing of packet flows. Secondly, the tasks running on the NFV platform may have heterogeneous processing requirements that OS schedulers (unlike hardware switches) address using their typical fair scheduling mechanisms. OS schedulers, however, do not treat packet flows fairly in proportion to their arrival rate. Thus, NF processing requires a re-thinking of the system resource management framework to address both these requirements. Moreover, standard OS schedulers: a) do not have the right metrics and primitives to ensure fairness between NFs that deal with the same or different packet flows; and b) do not make scheduling decisions that account for chain level information. If the scheduler allocates more processing to an upstream NF and the downstream NF becomes overloaded, packets are dropped by the downstream NF. This results in inefficient processing and wasting of the work done by the upstream NF. OS schedulers also need to be adapted to work with user space data plane frameworks such as Intel's DPDK [45]. They have to be cognizant of NUMA (Non-uniform Memory Access) concerns of NF processing, core affinity of an NF, and the dependencies among NFs in a service chain. Determining how to dynamically schedule NFs is key to achieving high performance and scalability for diverse service chains, especially in a scenario where multiple NFs are contending for a CPU core[16].

Hardware routers and switches that employ sophisticated scheduling algorithms such as rate proportional scheduling [59,60] have predictable performance per-packet because processing resources are allocated fairly to meet *Quality of Service* (QoS) requirements and bottlenecks are avoided by design. However, NFV platforms are necessarily different because:

a) the OS scheduler does not know a priori, the capacity or processing requirements of an NF or chain of NFs;
b) an NF may have variable per-packet costs (*e.g.,* some packets may trigger DNS lookup, which is expensive to process, and others may just be an inexpensive header match).

[16]While CPU core counts are increasing in modern hardware, they are likely to remain a bottleneck resource, especially when service chains are densely packed into a single machine (as is often the case with several proposed approaches [43, 58]).

With NFV service chains, there is a need to be aware of the computational demands for packet processing. There can also be sporadic blocking of NFs due to I/O (read/write) stalls.

A further consideration is that routers and switches 'simply' drop packets when congested. However, an NF in a service chain that drops packets can result in considerable wasted processing at NFs earlier in the chain. These wasted resources could be gainfully utilized by other NFs being scheduled on the same CPU core to process other packet flows.

3.2 System-level challenges with the deployment of Network Functions and Network Service Chaining

The middleboxes that are being deployed in the industry are diverse in their applications as well as in their complexity and processing requirements. ETSI standards [15] show that NFs have dramatically different processing and performance requirements. Measurements of existing NFs show the variation in CPU demand and per packet latency: some NFs have per-core throughput in the order of million packets per second (Mpps), *e.g.,* switches; others have throughputs as low as a few kilo pps, *e.g.,* encryption engines. Table 3.1 indicates the approximate per packet computation cost (CPU cycles) for some of the NFs[17] [61].

3.2.1 Diversity, Fairness, and Chain Efficiency

***#1* Fair Scheduling:** Despite this NF diversity in terms of processing requirements, determining how to allocate CPU time to all the contending NFs in order to provide fair and efficient chain performance is the focus of our work. Defining "Fairness" when NFs may have completely different requirements or behavior can be difficult. A measure of fairness that we leverage is the work on Rate Proportional Servers [59, 60], that apportion resources (CPU cycles) to NFs based on the combination of an NF's arrival rate and its processing cost.

Intuitively, if either one of these factors is fixed, then we expect its CPU allocation to be proportional to the other metric. For example, if two NFs have the same computation cost but one has twice the arrival rate, then we want it to have twice the output rate relative to the second NF. Alternatively, if the NFs have the same arrival

[17]Note: The reported numbers provide ballpark estimation on processing complexity of the corresponding middlebox; actual numbers are implementation specific and can vary with the type of platform, Operating System, libraries and language of implementation.

Table 3.1: Per Packet Processing cost in CPU computation cycles for different NFs.

Network Function	Compute Cycles/Packet
L2 Forwarding	~ 70
IP Routing	~ 175
L2-4 Classification	~ 750
TCP Termination	~ 1500
Stateful Firewall	~ 2250
OpenFlow Process	~ 5000
IDS/IPS	~ 5000
NextGen Firewall	~ 8500
IPsec / SSL	~ 9500
Firewall + SSL	~ 18000

rate, but one requires twice the processing cost, then we expect the heavy NF to get about twice as much CPU time, resulting in both NFs having the same output rate. This definition of fairness can of course be supplemented with a prioritization factor, allowing an understandable and consistent way to provide differentiated service for NFs.

Unfortunately, standard CPU schedulers do not have sufficient information to allocate resources in a way that provides rate-cost proportional fairness. CPU schedulers typically try to provide a fair allocation of processing time, but if computation costs vary between NFs this cannot provide rate-cost fairness. Therefore,

> The solution must seek to enhance the scheduler with additional information so that it can appropriately allocate CPU time to the contending NFs and provide a correctly weighted allocation of run-time.

***#2* Efficient Chaining:** Beyond simply allocating CPU time fairly to NFs on a single core, the combination of NFs into service chains demands careful resource management across the chain to minimize the impact of bottlenecks. Processing a packet only to have it dropped from a subsequent bottleneck's queue is wasteful, and a recipe for *receive livelock* [42, 62].

When an NF (whether a single NF or one in a service chain) is overloaded, packet drops become inevitable, and processing resources already consumed by those packets are wasted. For responsive flows, such as a TCP flow, congestion control and avoidance using packet drop methods such as *Random Early Drop* (RED), *Random Early Marking* (REM), *Stateless Fair Queuing* (SFQ), *Core Stateless Fair Queuing* (CSFQ) [63–66] and feedback with *Explicit Congestion Notification* (ECN) [67] can cause the flows to adapt their rates to the available capacity on an end-to-end basis. However, for non-responsive flows (e.g., UDP), a local, rapidly adapting method is backpressure, which can propagate information regarding a congested resource upstream (i.e., to previous NFs in the chain). Therefore,

> The solution must allow the upstream NFs and upstream nodes to account for the congestion at the downstream NFs and determine either to propagate the backpressure information further upstream or drop packets to avoid downstream congestion and to minimize the wasted work.

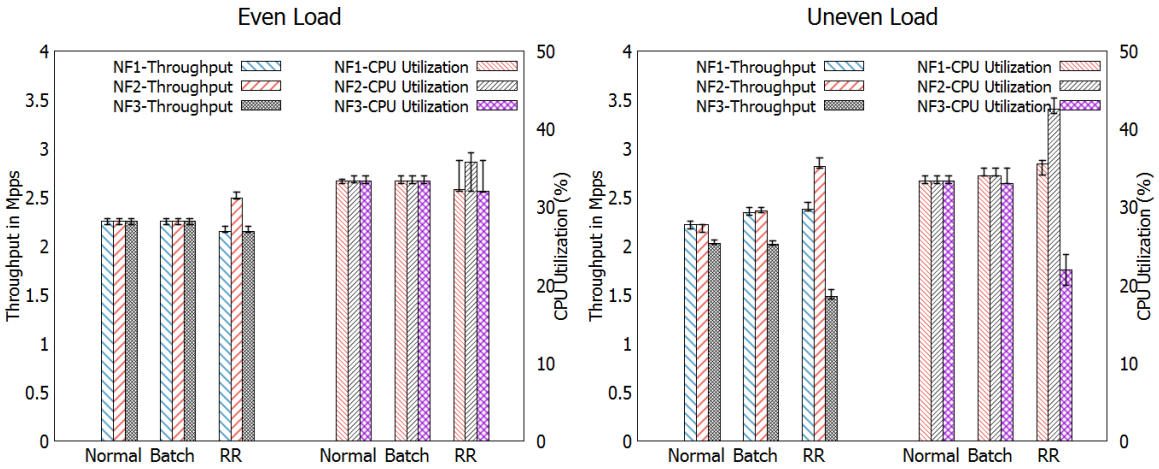
However, It is important to ensure that effects such as head-of-the-line blocking or unfairness do not creep in as a result of such backpressure notification.

3.2.2 Are existing OS schedulers well-suited for NFV deployment?

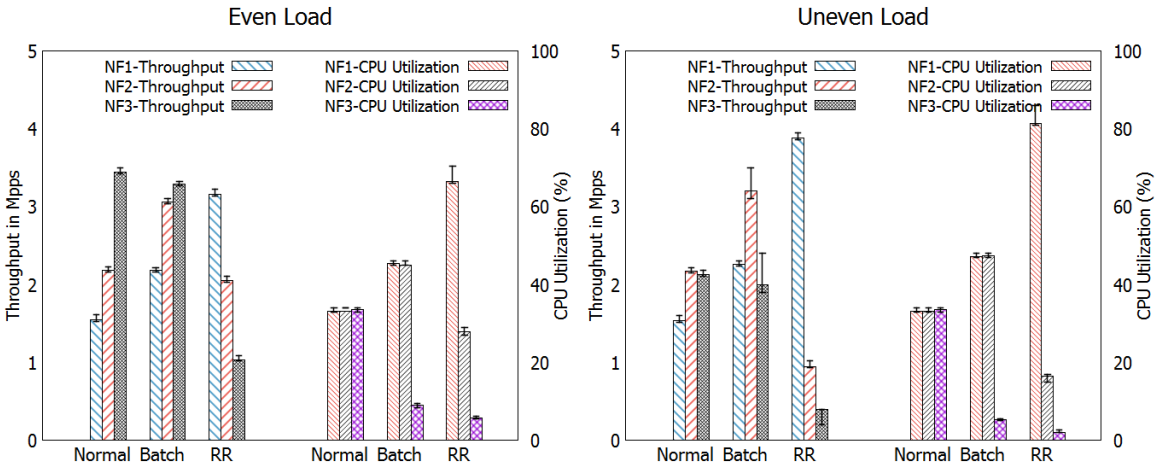
As discussed in section §2.5, Linux provides several different process schedulers, with the *Completely Fair Scheduler* (CFS) [54] being the default since kernel 2.6.23. In this work we focus on three different schedulers: i) CFS Normal, ii) CFS Batch, and *Round Robin* (RR)[18].

To explore the impact of these schedulers on NFV applications we consider a simple deployment with three NF processes sharing a CPU core. The NFs run atop a DPDK-based NFV platform that efficiently delivers packets to the NFs. We look at two workloads: 1) equal offered load to all NFs of 5 Mpps; 2) unequal offered load, with NF1 and NF2 getting 6 Mpps, and NF3 getting 3 Mpps. We also consider the case where NFs have different computation costs. As described above, the desirable behavior is for NFs to be allocated resources in proportion to both their arrival rate and processing requirements.

[18]Note: We do not consider the FIFO scheduler, which is another variant of real time scheduling class. Our evaluation for yield based NFs show FIFO to have similar characteristics and fare worse than RoundRobin due to unbounded starvation.



(a) Throughput for Homogeneous NFs (*i.e.*, NFs with same per packet computation cost).



(b) Throughput for Heterogeneous NFs (with different per packet computation cost)

Figure 3.1: The scheduler alone is unable to provide fair resource allocations that account for processing cost and load.
Left (Even Load): corresponds to equal offered load (packet arrival rate) on all NFs
Right (Uneven Load): corresponds to unequal variation in the offered load on all NFs.

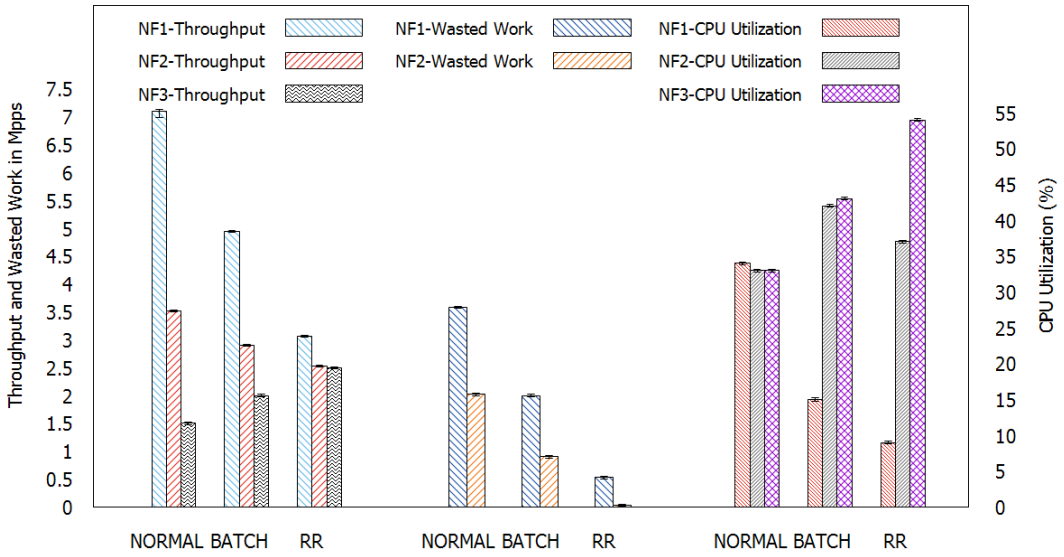


Figure 3.2: Throughput, wasted work and CPU utilization for 3NF chain sequence(NF1, NF2, NF3) subject to uniform load.

Table 3.2: Context Switches for Homogeneous NFs

	Even Load						Uneven Load					
	SCHED_ NORMAL		SCHED_ BATCH		SCHED_ RR		SCHED_ NORMAL		SCHED_ BATCH		SCHED_ RR	
NF	csw-ch/s	nvc swch /s	csw-ch/s	nv cswch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s
NF1	0	339	0	333	266	3	0	3544	0	527	247	5
NF2	0	334	0	333	265	4	0	6205	0	479	246	5
NF3	0	333	0	334	266	3	9753	9	1007	0	248	3

Table 3.3: Context Switches for Heterogeneous NFs

	Even Load						Uneven Load					
	SCHED_ NORMAL		SCHED_ BATCH		SCHED_ RR		SCHED_ NORMAL		SCHED_ BATCH		SCHED_ RR	
NF	csw-ch/s	nvc swch /s	csw-ch/s	nv cswch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s	csw-ch/s	nvc swch /s
NF1	0	33785	0	504	198	7	0	38585	0	503	85	10
NF2	0	32214	1	505	204	2	0	41089	4	496	92	1
NF3	65796	107	1010	8	206	0	79479	85	1004	4	93	0

3.2.2.1 NFs with Homogeneous Processing cost

In our first test, illustrated in Figure 3.1a, all 3 NFs have equal computation cost (roughly 250 CPU cycles per packet). With an even load sent to all NFs, we find that the three schedulers perform about the same, with an equal division of CPU time leading to equal throughputs for each NF. However, reducing the traffic to NF3 by half shows the different behavior of the schedulers: while the CFS-based schedulers continue to evenly divide the CPU (CFS's definition of fairness), the RR scheduler allocates CPU time in proportion to the arrival rate, which better matches our notion of rate proportional fairness.

This happens because RR uses a time quantum that is substantially longer than an NF ever needs, so NFs which yield the CPU earlier (*i.e.,* because they have fewer packets to process) receive less CPU time and thus have lower throughput. Note the context switches (shown in Table 3.2) in RR case are predominantly voluntary context switches, while the CFS based schedulers incur non-voluntary context switches.

3.2.2.2 NFs with Heterogeneous Processing cost

We next consider heterogeneous NFs (computation costs: NF1= 500, NF2=250 and NF3=50 CPU cycles) with even or uneven load. Figure 3.1b shows that when arrival rates are the same, none of the schedulers are able to provide our fairness goal—an equal output rate for all three NFs. CFS Normal always apportions CPU equally, regardless of offered load and NF processing cost, so the lighter weight NF3 gets the highest throughput. The RR scheduler is the opposite since it gives each NF an equal chance to run, but does not limit the time the NF runs for. The CFS Batch scheduler is in between these extremes since it seeks to provide fairness, but over longer time periods.

Notably, the Batch scheduler provides NF3 almost the same throughput as Normal CFS, despite allocating it substantially less CPU. The reason for this is that Normal CFS can incur a very large number of context switches due to its goal of providing very fine-grained fairness. Since Batch mode reduces scheduler preemption, it has substantially fewer non-voluntary context switches—reducing from 65K to 1K per second—as illustrated in Table 3.3. While RR also has low context switch overhead, it allows heavy weight NFs to greedily consume the CPU, nearly starving NF3.

3.2.2.3 NF Chain

We consider, a simple 3NF chain (NF1, NF2, and NF3, with computation costs: NF1=120, NF2=270 and NF3= 550 cycles), and subject the chain to a uniform load of 10Mpps. We profile the overall chain and per NF throughput, and also profile the default scheduler behavior for CFS, BATCH, and RR schedulers. Figure 3.2 shows the variation of CPU allocation time to different NFs and correspondingly impacting the throughput of the overall chain as well as the processing of distinct NFs. We account the wasted work for each of the upstream NFs in the chain as the number of packets processed by the upstream NF but dropped by the immediate downstream NF due to overflow on receive queue.

We can observe that CFS Normal always apportions CPU equally, regardless the NF processing cost, so the light weight NF1 gets the highest throughput, while NF2 and NF3 with the same time-share are not able to cope with the load of their immediate upstream NFs. On the other hand, RR and Batch schedulers apportion each NF an equal chance to run but do not limit the time the NF runs for. If the NFs voluntarily yield before their time-slice the other NFs get a chance and share the CPU. We can observe the NF1 to get the least share of CPU, while NF3 gets the highest CPU share. However, we can still observe significant wasted work across different schedulers. This is due to the lack of awareness of the processing cost of NFs and processing load on each of the intermediate NFs in the chain.

These results show that just having the Linux scheduler handle scheduling NFs has undesirable results as by itself it is unable to adapt to both varying per-packet processing requirements of NFs and packet arrival rates. Moreover, it is important to avoid the overheads of excessive context switches. All of these scheduling requirements must be met on a per-core basis while accounting for the behavior of chains spanning multiple cores or servers.

We posit that a scheduling framework for NF service chains has to simultaneously account for both task level scheduling on processing cores and packet level scheduling within an NF. This combined problem is what poses a challenge: *When you get a packet, you have to decide which task has to run, and also which packets to process, and for how long.* In addition, the chain-wide processing and load requirements need to be accounted to avoid any wasted-work.

3.2.3 Facilitating I/O for NFs

Next, we consider NFs that frequently perform I/O operations. With Linux, there are two traditional approaches for performing I/O *i.e.,* i) Synchronous mode and ii)

Asynchronous mode. We compare the performance impact of performing frequent I/O operation both on NF packet processing and overall throughput. Table 3.4 demonstrates the overall performance metrics of employing the synchronous and asynchronous mode of I/O.

Table 3.4: Synchronous vs Asynchronous I/O for 10MB HTTP Download and packet-logger NF

	Synchrnous I/O	Asynchronous I/O
Flow Completion Time (ms)	70-86	39-43
Throughput (Gbps)	0.93-1.14	1.87-2.06

The results clearly indicate the benefit of employing Asynchronous I/O for NFIs, however performing Asynchronous I/O in Linux is complex for the following reasons:

i) NFIs must explicitly reserve and manage the 'asynchronous I/O control blocks' to perform asynchronous read and write operations.

ii) NFIs need to select and register for desired asynchronous notification method that enables the NFIs to be notified of completion of the I/O operation in a variety of ways: a) by delivery of a signal, b) by instantiation of a thread, or c) no notification at all, but poll in user space.

iii) Most importantly, to ensure the correctness of operation (preserving the packet ordering) and to avoid *head-of-line* blocking, the NFIs need to synchronize their packet processing with asynchronous I/O completion notification, along with co-ordination of voluntarily yield decisions on CPU.

These set of operations though common across different NFIs, demand calculated tuning and configuration of the asynchronous I/O parameters. Hence,

A general abstraction framework in EMS would be more desirable that can effectively manage and co-ordinate with NF scheduling to determine when to wake-up and when to relinquish the NF from packet processing.

Chapter 4

Related Work

In this Chapter, we present the literature survey on the state-of-the-art work in the prospect of a) High Performance NFV platforms and scheduling control for NFs and b) Queue management, specifically the congestion control, ECN and backpressure schemes in SFCs.

Contents

4.1 High Performance NFV Platforms and Scheduling of Network Functions

In recent years, several NFV platforms have been developed to accelerate packet processing on commodity servers [43, 44, 47, 68, 69]. There is a growing interest in managing and scheduling network functions. Many works address the placement of middleboxes and NFs for performance target or efficient resource usage [3,22,70–73]. For example, E2 [22] builds a scalable scheduling framework on top of BESS [69]. They abstract NF placement as a DAG, dynamically scale and migrate NFs while keeping flow affinity. NFV-RT [73] defines deadlines for requests, and places or migrates NFs to provide timing guarantees. These projects focus on NF management and scheduling across the cluster scale.

Also, several works have addressed high performance for NFV platforms by developing network applications on specialized networking hardware such as *Network Processor Unit*s (NPUs) [74], *Field-Programmable Gate Array*s (FPGAs) based programmable switches [75, 76], and most recently have even considered *Graphics Processing Unit*s (GPUs) to accelerate packet processing for NFV [77]. The works P4 [76] and Packet Transactions [78] have looked at providing high level programming tools for such hardware.

The focus of our work is on a different scale: *i.e.,* how to efficiently schedule the NFs on shared cores in order to achieve fairness and maximize the chain-wide throughput by avoiding the wasted-work across NF chains. The diversity of software-based NFs or the middleboxes, coupled with varied nature of I/O and packet processing costs make the scheduling of NFs more complex and different from traditional packet scheduling for fairness on hardware platforms [60, 79–81]. Furthermore, different kinds of flow arrival rates exacerbate the difficulty of fair scheduling.

PSPAT [82] is a recent host-only software packet scheduler. PSPAT aims to provide a scalable scheduler framework by decoupling the packet scheduler algorithm from dispatching packets to the NIC for high performance.

NFVnice considers the orthogonal problem of packet processing cost and flow arrival rate to fairly allocate CPU resources across the NFs.

PIFO [83] presents the packet-in-first-out philosophy distinct from the typical first-in-first-out packet processing models. In this model, right at the time of packet arrival, the decision on whether to accept a packet and queue it for processing at the intended NF or discard the packet is taken. Then enqueued packets are always processed in order. This approach of selective early discard yields two benefits:

i) it avoids dropping partially processed (through the chain) packets, thus not wasting CPU cycles;

ii) it avoids CPU stealing and allows CPU cycles to be judiciously allocated to the other contending NFs.

We use the insight from this work to decide whether to accept a packet and queue it for processing at the intended NF or discard at the time of packet arrival.

4.2 User space scheduling and related frameworks

Works, such as [84, 85], consider cooperative user-space scheduling, providing very low cost context switching, that is orders of magnitude faster than regular Pthreads. However, the drawbacks with such a framework are two-fold:

a) They invariably require the threads to cooperate, i.e., each thread must voluntarily yield to ensure that the other threads get a chance to share the CPU, without which progress of the threads cannot be guaranteed. This means that the programs that implement L-threads must include frequent rescheduling points for each L-thread [85] incurring additional complexity in developing the NFs.

b) As there is no specific scheduling policy (it is just FIFO based), all the L-threads share the same priority, and are backed by the same kernel thread (typically pinned to a single core), and thus lack the ability to perform selective prioritization and the ability to provide QoS differentiation across cooperating threads.

Another approach used by systems such as E2 [22] and VPP [68] is to host multiple NFs within a shared address space, allowing them to be executed as function calls in a run to completion manner by one thread. This incurs very low NUMA and cross-core packet chaining overheads, but being monolithic, it is inflexible and impedes the deployment of NFs from third party vendors.

4.3 Queue Management: Congestion Control and Backpressure

Congestion control and backpressure have been extensively studied in the past [86–91]. DCTCP [86] leverages ECN to provide multi-bit feedback to the end hosts. MQ-ECN [88] enables ECN for tradeoff of both high throughput and low latency in multi-service multi-queue production DCNs(Data Center Network). All of these focus on congestion control in DCNs.

However, in an NFV environment, flows are typically steered through a chain of network functions. This implies that more the later the congestion is found, the more resources are wasted in the upstream of the chain. If the end hosts do not enable ECN support or there are *User Datagram Protocol* (UDP) flows, it is especially important for the NFV platform to gracefully handle high load scenarios in an efficient and fair way.

4.4 Fair sharing of resources

Fair sharing has been extensively studied under network packet schedules, *Operating System* (OS) process schedulers and job schedulers in Cloud.

In the context of packet schedulers, the concept of fair queuing corresponds to the logic of selecting the next packet from the pool of multiple queued packets for transmission over the wire (link). The general concept is to keep track of packets scheduled for transmission across all the backlogged flows and update the scheduling decision to pick the packet from least serviced flow. For example, Weighted Fair Queueing (WFQ) [52], Start-time Fair Queueing (SFQ) [80], Generalized Processor Sharing (GPS) [59], and Deficit Round Robin (DRR) [79]. Notably, the single-link schedulers such as WFQ [52] and WF2Q [92] track virtual runtime to account for flows share and require the knowledge of packet size to schedule the packets as they order the flows based on their finish tag. However, SFQ [80] does not need the packet size information a priori but computes the start tag based on the transmission time of the previous packets obtained at the end of transmission.

In the context of OS schedulers, the concept is quite analogous to packet schedulers, but applies to the selection of next task or a process from the pool of run queue (all the waiting tasks) for execution on a CPU core. In addition, the scheduler also needs to determine the quantum (time-slice) that a task needs to spend on a CPU core to guarantee the fair share. Distributed Weighted Round Robin scheduler (DWRR) [93] achieves a scalable proportionally fair scheduling by adapting the time slice of the round in proportion to the weight of the scheduled task. The CFS scheduler is analogous to weighted fair queuing (WFQ) scheduling [52, 53]. Thus, CFS ensures a fair proportion of CPU allocation to all the tasks.

Work such as [94, 95], propose to ensure fair sharing of network resources among multiple tenants by spreading requests to multiple processing entities. That is, they distribute flows with different costs to different processing threads. [94] accounts for the virtual time to measure the processing time accounted by each flow on their dominant resource. Based on this metric, the packets of the flow are scheduled to ensure max-min fairness. Thus it accounts to seek dominant resource fairness for all the contending flows. [95] separates the requests with different costs/size and assigns them to different worker threads. Thus, even with bursty workloads it ensures fair workload distribution and achieves multi-tenant isolation.

In contrast, we seek to achieve fairness by scheduling the NFs that process the packets of different flows appropriately, Thus, a fair share of the CPU is allocated to each competing NF.

Chapter 5

High Performance Network Function Chains

In this section, we present our NF management and scheduling framework and outline our design choices, implementation details, system parameter tuning, testbed setup and evaluation to quantify the associated benefits and overheads of our proposal.

Contents

5.1 Introduction

To resolve the challenges outlined in §3.2 we propose NFVnice, an NFV management framework that provides fair and efficient resource allocations to NF service chains. NFVnice focuses on the scheduling and control problems of NFs running on shared CPU cores, and considers a variety of realistic issues such as bottlenecked NFs in a chain, and the impact of NFs that perform disk I/O accesses, which naturally complicate scheduling decisions. NFVnice makes the following contributions:

- Rate-Cost proportional Scheduling: We introduce the basic notion of fairness that integrates both the notion of fairness from Hardware schedulers and the OS CPU schedulers.
- Automatically tuning CPU scheduling parameters in order to provide a fair allocation that weighs NFs based on both their packet arrival rate and the required computation cost.
- Determining when NFs are eligible to get a CPU share and when they need to yield the CPU, entirely from user space, improving throughput and fairness regardless of the kernel scheduler being used.
- Leveraging the scheduling flexibility to achieve backpressure for service chain-level congestion control, that avoids unnecessary packet processing early in a chain if the packet might be dropped later on.
- Extending backpressure to apply not only to adjacent NFs in a service chain but for full service chains and managing congestion across hosts using ECN.
- Presenting a userspace OS scheduler-agnostic framework that does not require any operating system or kernel modifications.
- Presenting efficient I/O management abstraction to NFs by leveraging the Linux 'Asynchronous I/O' APIs.

We have implemented NFVnice on top of OpenNetVM [46], a DPDK based NFV platform that runs NFs in separate processes or docker containers.

5.2 Design Choices, Architecture and Design

In an NFV platform, at the top of the stack are one or more network functions that must be scheduled in such a way that idle work (i.e., while waiting for packets)

is minimized and load on the service chain is shed *as early as possible* so as to avoid wasted work. However, the operating system's process scheduler that lies at the bottom of the software stack remains completely application agnostic, with its goal of providing a fair share of system resources to all processes. As shown in earlier §3.2, the kernel scheduler's metrics for scheduling are along orthogonal dimensions to those desired by the network functions. NFVnice bridges the gap by translating the scheduling requirements at the NFV application layer to a format consumable by the operating system.

The design of NFVnice centers around the concept of assisted preemptive scheduling, where network functions provide hints to the underlying OS with regard to their utilization. In addition to monitoring the average computation time of a network function per packet, NFVnice needs to know when NFs in a chain are overloaded or blocked on packet/disk I/O. The queues between NFs in a service chain serve as a good indicator of pending work at each NF. To facilitate the process of providing these metrics from the NF implementation to the underlying operating system, NFVnice provides network function implementations with an abstraction library called *libnf*. In addition to the usual tasks such as efficient reading/writing packets from/to the network at line rate and overlapping processing with non-blocking asynchronous I/O, *libnf* co-ordinates with the NFVnice platform to schedule/de-schedule a network function as necessary.

Modifying the OS scheduler to be aware of various queues in the NFV platform is an onerous task that might lead to unnecessary maintenance overhead and potential system instability. One approach is to change the priority of the NF based on the queue length of packet at that NF. This will have the effect of increasing the number of CPU cycles provided to that NF. This will require the change to occur frequently as the queue length varies. The change requires a system call, which consumes CPU cycles and adds latency. In addition, with service chains, as the queue at an upstream NF builds, its priority has to be raised to process packets and deliver to a queue at the downstream NF. Then, the downstream NF's priority will have to be raised. We believe that this can lead to instability because of frequent changes and the delay involved in effecting the change. This only gets worse with complex service chains, where an NF is both an upstream NF for one service chain and a downstream NF for another service chain. Instead, NFVnice leverages cgroups [55, 56], a standard user space primitive provided by the operating system to manipulate process scheduling. NFVnice monitors queue sizes, computation times and I/O activities in user space with the help of *libnf* and manipulates scheduling weights accordingly.

5.2.1 Rate-Cost Proportional Fair Scheduling

We present novel rate-cost proportional fairness for NF chains. By rate-cost proportional we account both the network characteristic of packet-arrival rate and the system characteristic of computation cost the packets at the NF. The details on estimating the associated scheduling parameters and the methodology is illustrated in section §5.2.3. We adopt the notion of rate-cost proportional fairness for two fundamental reasons:

i) it not only seeks to maximize the throughput for a given load across NFs, but even in the worst case scenarios (highly uneven and high overload across competing NFs), it ensures that all competing NFs get a minimal CPU share necessary to progress the NFs; and

ii) the rate-cost proportional fairness is general and flexible, so that it can be tuned to meet the QoS policies desired by the operator.

Further, the approach ensures that when contending NFs include malicious NFs (those that fail to yield), or misbehaving NFs (get stuck in a loop making no progress), such NFs do not consume the CPU excessively, impeding the progress of other NFs. While the Linux default scheduler addresses this through the notion of a virtual run-time for each running task, we fine-tune that capability to provide the correct share of the CPU for an NF, rather than just allocating an equal share of the CPU for each contending NF.

5.2.2 System Components

Figure 5.1 illustrates the key components of the NFVnice platform. We leverage DPDK for fast user space networking [45]. Our NFV platform is implemented as a system of queues that hold packet descriptors pointing to shared memory regions. The NF Manager runs on a dedicated set of cores and is responsible for ferrying packet references between the NIC queues and NF queues in an efficient manner. When packets arrive to the NIC, Rx threads in the NF Manager take advantage of DPDK's poll mode driver to deliver the packets into a shared memory region accessible to all the NFs. The Rx thread does a lookup in the Flow Table to direct the packet to the appropriate NF. Once a flow is matched to an NF, packet descriptors are copied into the NF's receive ring buffer and the Wakeup subsystem brings the NF process into the runnable state. After being processed by an NF, the NF Manager's Tx Threads move packets through the remainder of the chain. This provides zero-copy packet movement.

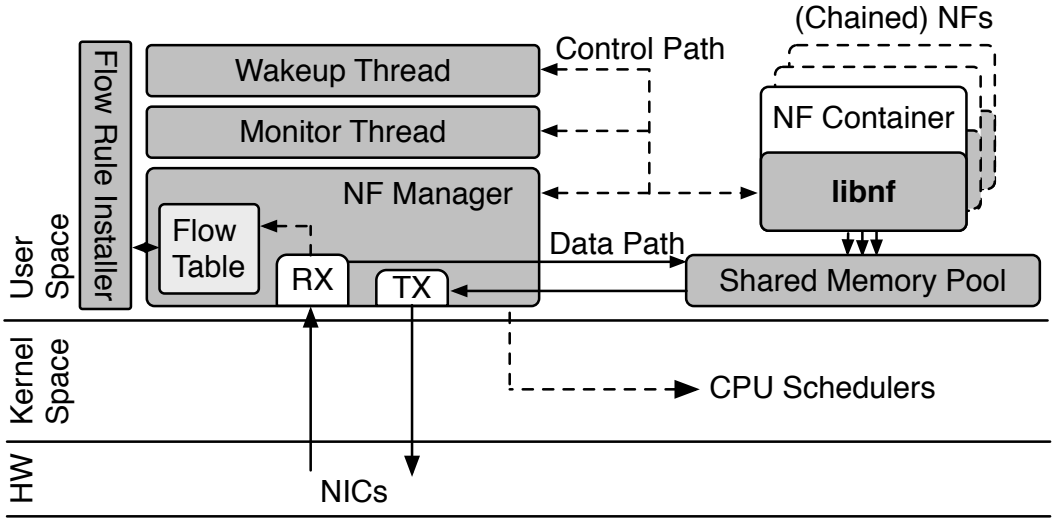


Figure 5.1: NFVnice Building Blocks

Service chains can be configured during system startup using simple configuration files or from an external orchestrator such as an SDN controller. When an NF finishes with a packet, it enqueues it in its Tx queue, where it is read by the manager and redirected to the Rx queue of the next NF in the chain. The NF Manager also picks up packets from the Tx queue of the last NF in the chain, and sends it out over the network.

We have designed NFVnice to provide high performance processing of NF service chains. The NF Manager's scheduling subsystem determines when an NF should be active and how much CPU time it should be allocated relative to other NFs. The backpressure subsystem provides chain-aware management, preventing NFs from spending time processing packets that are likely to be dropped downstream. Finally, the I/O interface facilitates efficient asynchronous storage access for NFs.

5.2.3 Scheduling NFs

Each network function in NFVnice is implemented inside its own process (potentially running in a container). Thus the OS scheduler is responsible for picking which NF to run at any point in time. We believe that rather than design an entirely new scheduler for NFV, it is important to leverage Linux's existing scheduling framework, and use our management framework in user space to tune any of the stock OS schedulers to provide the properties desired for NFV support. In particular, we exploit the CFS Batch scheduler, but NFVnice provides substantially similar ben-

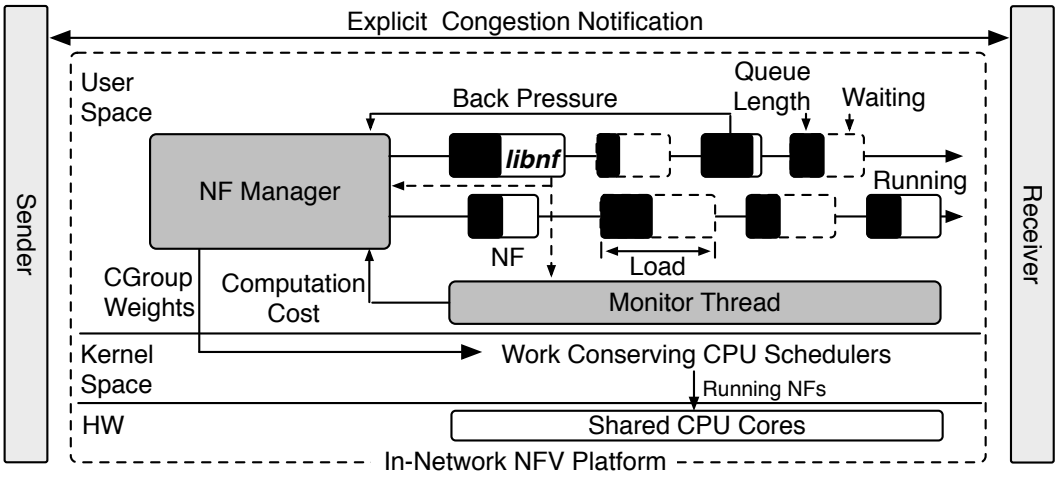


Figure 5.2: NF Scheduling and Backpressure

efits to each of the other Linux kernel schedulers. Figure 5.2 shows the NFVnice scheduling that makes the OS scheduler be governed by NF Manager via cgroups, and ultimately assigns running NFs to shared CPU cores. The detailed description of the figure is in the Sections 5.2.3 and 5.2.4.

Activating NFs: NFs that busy wait for packets perform very poorly in a shared CPU environment. Thus it is critical to design the NF framework so that NFs are only activated when there are packets available for them to process, as is done in NFV platforms such as netmap [44] and ClickOS [47]. However, these systems provide only a relatively simple policy for activating an NF: once one or more packets are available, a signal is sent to the NF so that it will be scheduled to run by the OS scheduler in netmap, or the hypervisor scheduler in ClickOS. While this provides an efficient mechanism for waking NFs, neither system allows for more complex resource management policies, which can lead to unfair CPU allocations across NFs, or inefficient scheduling across chains.

In NFVnice, NFs sleep by blocking on a semaphore shared with the NF Manager, granting the management plane great flexibility in deciding which NFs to activate at a given time. The policy we provide for activating an NF considers the number of packets pending in its queue, its priority relative to other NFs, and knowledge of the queue lengths of downstream NFs in the same chain. This allows the management framework to indirectly affect the CPU scheduling of NFs to be fairness and service-chain aware, without requiring that information be synchronized with the kernel's scheduler.

Relinquishing the CPU: NFs process batches of packets, deciding whether to keep processing or relinquish the CPU between each batch. This decision and all interactions with the management layer, e.g., to receive a batch of packets, are mediated by *libnf*, which in turn exposes a simple interface to developers to write their network function. After a batch of at most 32 packets is processed, *libnf* will check a shared memory flag set by the NF Manager that indicates if it should relinquish the CPU early (e.g., as a result of backpressure, as described below). If the flag is not set, the NF will attempt to process another batch; if the flag has been set or there are no packets available, the NF will block on the semaphore until notified by the Manager. This provides a flexible way for the manager to indicate that an NF should give up the CPU without requiring the kernel's CPU scheduler to be NF-aware.

CPU Scheduler: Since multiple NF processes are likely to be in the runnable state at the same time, it is the operating system's CPU scheduler that must determine which to run and for how long. In the early stages of our work we sought to design a custom CPU scheduler that would incorporate NF information such as queue lengths into its scheduling decisions. However, we found that synchronizing queue length information with the kernel, at the frequency necessary for NF scheduling, incurred overheads that outweighed any benefits.

Linux's CFS Batch scheduler is typically used for long running computationally intensive tasks because it incurs fewer context switches than standard CFS. Since NFVnice carefully controls when individual NF processes are runnable and when they yield the CPU (as described above), the batch scheduler's longer time quantum and less frequent preemption are desirable. In most cases, NFVnice NFs relinquish the CPU due to policies controlled by the manager, rather than through an involuntary context switch. This reduces overhead and helps NFVnice prioritize the most important NF for processing without requiring information sharing between user and kernel space.

Assigning CPU Weights: NFVnice provides mechanisms to monitor a network function to estimate its CPU requirements, and to adjust its scheduling weight. Policies in the NF Manager can then dynamically tune the scheduling weights assigned to each process in order to meet operator specified priority requirements.

The packet arrival rate for a given NF can be easily estimated by either the NF or the NF Manager. We measure the service time to process a packet inside each NF using *libnf*. To avoid outliers from skewing these measurements (e.g., if a context switch occurs in the middle of processing a packet), we maintain a histogram of timings, allowing NFVnice to efficiently estimate the service time at different

percentiles.

$$load(NF_i) = (\lambda_i * s_i) + (n_{qi} * s_i) \tag{5.2.1}$$

$$TotalLoad(m) = \sum_{i=1}^{n} load(NF_i) \tag{5.2.2}$$

For each NF NF_i on a shared core m, we calculate the load on the NF as shown in Eq. 5.2.1 *i.e.,* the sum of the product of arrival rate, λ, and per packet service time, s_i and the product of backlog packets of NF n_{qi}[19] and per packet service time, s_i. We then find the total load on each core *e.g.,* for core m as shown in Eq 5.2.2, and assign cpu shares for NF_i on $core_m$ following the formula:

$$Shares_i = Priority(NF_i) * \frac{load(NF_i)}{TotalLoad(m)} \tag{5.2.3}$$

This provides an allocation of CPU weights that provides rate proportional fairness to each NF. The $Priority(NF_i)$ parameter can be tuned if desired to provide differential service to NFs. Tuning priority in this way provides a more intuitive level of control than directly working with the CPU priorities exposed by the scheduler since it is normalized by the NF's load.

5.2.4 Backpressure

A key goal of NFVnice is to avoid wasting work, *i.e.,* preventing an upstream NF from processing packets if they are just going to be dropped at a downstream NF later in the chain that has become overloaded. We achieve this through backpressure, which ensures bottlenecks are quickly detected while minimizing the effects of head of line blocking.

Cross-Chain Pressure: The NF Manager is in an ideal position to observe behavior across NFs since it assists in moving packets between them. When one of the NF Manager's TX threads detects that the receive queue for an NF is above a high watermark (HIGH_WATER_MARK) and queuing time is above threshold, then it examines all packets in the NF's queue to determine what service chain they are a part of. NFVnice then enables *service chain-specific* packet dropping at the upstream NFs. NF Manager maintains states of each NF, and in this case, it moves

[19]is the number of packets waiting to be processed by NF in its Rx Queue

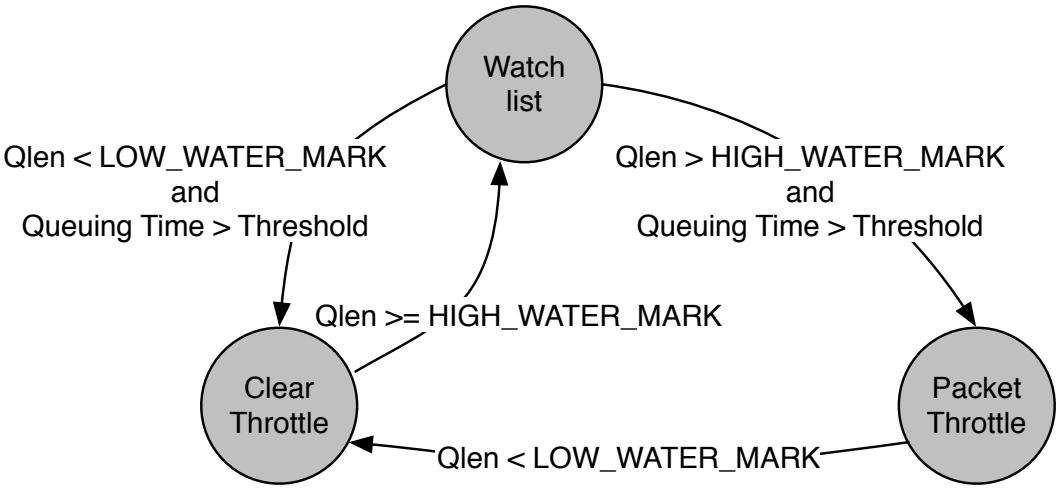


Figure 5.3: Backpressure State Diagram

the NF's state from *watch list* to *packet throttle* as shown in Figure 5.3. When the queue length becomes less than a low watermark (LOW_WATER_MARK), the state moves to *c*lear throttle, then again moves to the watch list if the queue length goes beyond the high mark.

The backpressure operation is illustrated in Figure 5.4, where four service chains (A-D) pass through several different NFs. The bold NFs (3 and 5) are currently overloaded. The NF Manager detects this and applies back pressure to flows A, C, and D. This is performed upstream where those flows first enter the system, minimizing wasted work. Note that backpressure is selective based on service chain, so packets for service chain B are not affected at all. Service chains can be defined at fine granularity (*e.g.,* at the flow-level) in order to minimize head of line blocking.

This form of system-wide backpressure offers a simple mechanism that can provide substantial performance benefits. The backpressure subsystem employs hysteresis control to prevent NFs rapidly switching between modes. Backpressure is enabled when the queue length exceeds a high watermark and is only disabled once it falls below the low watermark.

Local Optimization and ECN: NFVnice also supports simple, local backpressure, *i.e.,* an NF will block if its output TX queue becomes full. This can happen either because downstream NFs are slow, or because the NF Manager TX Thread responsible for the queue is overloaded. Local backpressure is entirely NF-driven, and requires no coordination with the manager, so we use it to handle short bursts

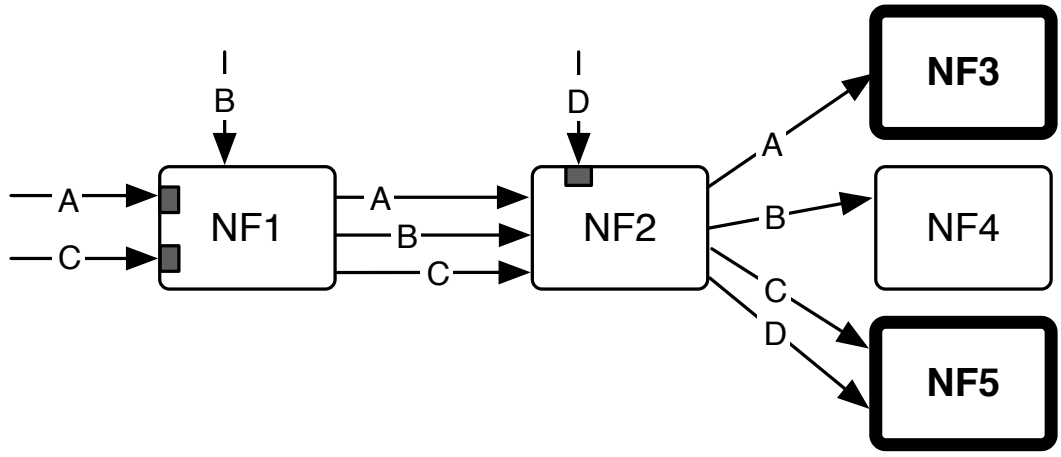


Figure 5.4: Overloaded NFs (in bold) cause back pressure at the entry points for service chains A, C, and D.

and cases where the manager is overloaded.

We also consider the fact that an NFVnice middlebox server might only be one in a chain spread across several hosts. To facilitate congestion control across machines, the NF Manager will also mark the ECN bits in TCP flows in order to facilitate end-to-end management. Since ECN works at longer timescales, we monitor queue lengths with an exponentially weighted moving average and use that to trigger marking of flows following [67].

5.2.5 Facilitating I/O

A network function could block when its receive ring buffer is empty or when it is waiting to complete I/O requests to the underlying storage. In both cases, NF implementations running on the NFVnice platform are expected to yield the CPU, returning any unused CPU cycles back to the scheduling pool. In case of I/O, NF implementations should use asynchronous I/O to overlap packet processing with background I/O to maintain throughput. NFVnice provides a simple library called *libnf* that abstracts such complexities from the NF implementation.

The *libnf* library exposes a simple set of APIs that allow the application code to read/write packets from the network, and read/write data from storage. The APIs are shown in Listing 15.3. If the receive ring buffer is empty while calling the `libnf_read_pkt` API, *libnf* notifies the NF manager and blocks the NF until further packets are available in the buffer.

```
// Read the next packet from the receive ring buffer
packet_descriptor* libnf_read_pkt();

// Output the processed packet to specified destination
int libnf_write_pkt(packet_descriptor*);

// Enqueue request to read from storage. Flow specific data can be stored
    in context
int libnf_read_data(int fd, void *buf,
      size_t size, size_t offset,
      void (*callback_fn)(void *), void *context);

// Enqueue request to write to storage. Flow specific data can be stored in
    context
int libnf_write_data(int fd, void *buf,
      size_t size, size_t offset,
      void (*callback_fn)(void *), void *context);
```

Figure 5.5: *libnf* API exposed to network function implementations.

In case of I/O, an NF implementation uses the `libnf_read_data` and `libnf_write_data` APIs. I/O requests can be queued along with a callback function that runs in a separate thread context. Using batched asynchronous I/O with double buffering, *libnf* enables the NF implementation to put the processing of one or more packets on hold, while continuing processing of other packets unhindered.

Batching reads and writes allows an NF to continue execution without waiting for I/O completion. The size of the batches and the flush interval is tunable by the NF implementation. Double buffering enables *libnf* to service one set of I/O requests asynchronously while the other buffer is filled up by the NF. When both buffers are full, *libnf* suspends the execution of the NF and yields the CPU.

5.2.6 System Management and NF deployment

The NF Manager 's (Rx, Tx and Monitor) threads are pinned to separate dedicated cores. The number of Rx, Tx and monitor threads are configurable (`C-Macros`), to meet system needs, and available CPU resources. Similarly, the maximum number of NFs and maximum chain length can be configured. NFVnice allows NFs and NF service chains to be deployed as independent processes or Docker containers which are linked with *libnf* library. *libnf* exports a simple, minimal interface (9 functions, 2 callbacks and 4 structures), and both the NF Manager and *libnf* leverage the DPDK libraries (ring buffers, timers, memory management). We believe developing

or porting NFs or existing docker containers can be reasonably straightforward. For example, a simple bridge NF or a basic monitor NF is less than 100 lines of `C` code.

5.3 System Implementation and Optimizations

We implemented NFVnice on top of OpenNetVM [46], which is a DPDK [45] based open-source NFV platform. OpenNetVM enables to deploy and run the NFs as either separate processes or as docker containers.

5.3.1 VNFM and EMS components

We implemented the following modules and extensions to the core NF Manager[20] and *libnf*[21] components of OpenNetVM [46]:

- **NF Wakeup Manager**: This module is implemented in NF Manager and runs on a dedicated CPU core. This module is responsible for identifying the NFs that are eligible to be scheduled and make them runnable and also to notify NFs to relinquish CPU when necessary.
- **NF Load Monitor**: This functionality is implemented in NF Manager and makes use of the existing monitor thread, which runs on a dedicated CPU core. It periodically monitors for the load on each NF and determines the cgroup weight of each NF on distinct CPU cores to account for right proportion of CPU share for the NFs.
- **Backpressure Monitor and Marking**: This functionality is implemented within the NF Manager. It is implemented as part of Rx and Tx threads, which move the packets from NIC ring descriptors to NF ring buffers identify the instantaneous queue length to enable or disable backpressure for any NF.
- ***libnf***: We extend the *libnf* to seamlessly switch the NF processing from poll mode to interrupt driven NFs, that can be woken up and de-scheduled as necessary. Additionally, we implemented light weight NF packet processing cost profiler that periodically monitors the per-packet processing cost and maintains the histogram of processing cost.

For evaluation purposes, we also implemented compute and I/O specific NFs like

[20] corresponds to the *Virtualized Network Function Manager* (VNFM) component in the ETSI NFV-MANO architecture.

[21] provides generic system abstractions for VNFI developers and corresponds to the *Element Management System* (EMS) component in the ETSI NFV-MANO architecture.

'basic forwarding', *'vlan tagger'*, *'packet and stream logger'* NFs that exhibit diverse computer and I/O costs.

5.3.2 Optimizations

Separating overload detection and control. Since the NFV platform [43] must process millions of packets per second to meet line rates, we separate out overload detection from the control mechanisms required to respond to it. The NF Manager's Tx threads are well situated to detect when an NF is becoming backlogged as it is their responsibility to enqueue new packets to each NF's Tx queue. Using a single DPDK's enqueue interface, the Tx thread enqueues a packet to a NF's Rx queue if the queue is below the high watermark, while getting feedback about the queue's state in the return value. When overload is detected, an overload flag is set in the meta data structure related to the NF.

The control decision to apply backpressure is delegated to th NF Manager's Wakeup thread. The Wakeup thread scans through the list of NFs classifying them into two categories: ones where backpressure should be applied and ones that need to be woken up. This separation simplifies the critical path in the Tx threads and also provides some hysteresis control, since a short burst of packets causing an NF to exceeds its threshold may have already been processed by the time the Wakeup thread considers it for backpressure.

Separating load estimation and CPU allocation. The load on an NF is a product of its packet arrival rate and the per-packet processing time. The scheduler weight is calculated based on the load and the cgroup's weights for the NF are updated. Since changing a weight requires writing to the Linux sysfs, it is critical that this be done outside of the packet processing data path. *libnf* merely collects samples of packet processing times, while the NF Manager computes the load and assigns the CPU shares using cgroup virtual file system.

The data plane (*libnf*) samples the packet processing time in a lightweight fashion every millisecond by observing the CPU cycle counter before and after the NF's packet handler function is called. We chose sampling because measuring overhead for each packet using the CPU cycle counters results in a CPU pipeline flush [96], resulting in additional overhead. The samples are stored in a histogram, in memory shared between *libnf* and the NF Manager.

The processing time samples produced by each NF are stored in shared memory and aggregated by the NF Manager. Not all packets incur the same processing time, as some might be higher due to I/O activity. Hence, NFVnice uses the median over

a 100ms moving window as the estimated packet processing time of the NF. Every millisecond, the NF Manager calculates the load on each NF using its packet arrival rate and the estimated processing time. Every 10ms, it updates the weights used by the kernel scheduler.

5.4 Evaluation

5.4.1 Testbed and Approach

Our experimental testbed has a small number of Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz servers, 157GB memory, running Ubuntu SMP Linux kernel 3.19.0-39-lowlatency. Each CPU has dual-sockets with a total of 56 cores. For these experiments, nodes were connected back-to-back with dual-port 10Gbps DPDK compatible NICs to avoid any switch overheads.

We make use of DPDK based high speed traffic generators, Moongen [97] and Pktgen [98] as well as Iperf3 [99], to generate line rate traffic consisting of UDP and TCP packets with varying numbers of flows. Moongen and Pktgen are configured to generate 64 byte packets at line rate (10Gbps), and vary the number of flows as needed for each experiment.

We demonstrate NFVnice's effectiveness as a user-space solution that influences the NF scheduling decisions of the native Linux kernel scheduling policies, *i.e.,* Round Robin (RR) for the Real-time scheduling class, SCHED_NORMAL (termed NORMAL henceforth) and SCHED_BATCH (termed BATCH) policies in the CFS class. Different NF configurations (compute, I/O) and service chains with varying workloads (traffic characteristics) are used. For all the bar plots, we provide the average, the minimum and maximum values observed across the samples collected every second during the experiment. In all cases, the NFs are interrupt driven, woken up by NF manager when the packets arrive while NFs voluntarily yield based on NFVnice's policies. Also, when the transmit ring out of an NF is full, that NF suspends processing packets until room is created on the transmit ring.

5.4.2 System parameter tuning and study of tradeoffs

5.4.2.1 Tuning NFVnice

The key parameters that need to be tuned for NFVnice are the marking thresholds and the interval for periodic state profiling.

Table 5.1: Packet drop rate per second

	NORMAL		BATCH		RR(1ms)		RR(100ms)	
	Default	NFVnice	Default	NFVnice	Default	NFVnice	Default	NFVnice
NF1	3.58M	11.2K	2M	0	0.86M	0	0.53M	0
NF2	2.02M	12.3K	0.9M	11.5K	2.92M	12K	0.03M	12K

Marking Thresholds: To tune the key parameters of NFVnice, viz., the HIGH_WATER_MARK and LOW_WATER_MARK, the thresholds for the queue occupancy in the Rx ring, we measure the throughput, wasted work, context-switch overheads and achieved Instructions per Cycle (IPC) count for different configurations. We use a 3 NF, "Low-Med-High" service chain, and use Pktgen to generate line rate minimum packet size traffic.

We begin with a fixed 'margin' (difference between the High and Low thresholds). With the margin at 30, we vary the high threshold. Below 70%, the throughput starts to drop (under-utilization), while above 80% the number of packet drops at the upstream NFs increases (insufficient buffering). We then varied the NF service chain length (from 2 to 6), and computation costs (per packet processing cost from 100 cycles to 10000 cycles) to see the impact of setting the water marks. Across all these cases, we observed that a choice of 80% for the HIGH_WATER_MARK worked 'well'. With the high water mark fixed at 80%, we varied the LOW_WATER_MARK, by varying the margin. With a very small margin (1 to 5), packet drops increased, while a margin above 30 degraded throughput. We chose a margin of 20 because it provided the best performance across these experiments.

We acknowledge that these watermark levels and thresholds are sensitive to overall path-delay, chain length and processing costs of the NFs in the chain, and that these parameters are necessarily an engineering compromise.

Periodic profiling and CPU weight assignment granularity: We based our frequency of CPU profiling based on the overheads of rdtsc (observed to be roughly 50 clock cycles) and average time to write to the cgroup virtual file system (5 μ seconds). We discard the first 10 samples to effectively account for warming the cache and to eliminate outliers.

5.4.3 Overall NFVnice Performance

We first demonstrate NFVnice's overall performance, both in throughput and in resource (CPU) utilization for each scheduler type. We compare the default schedulers to our complete NFVnice system, or when only including the CPU weight allocation tool (which we term `cgroups`) or the `backpressure` to avoid wasted work at

Table 5.2: Scheduling Latency and Runtime of NFs

	NORMAL		BATCH		RR(1ms)		RR(100ms)	
measured in ms	Default	NFVnice	Default	NFVnice	Default	NFVnice	Default	NFVnice
NF1-Avg. Delay	0.002	0.112	0.003	1.613	1.022	0.730	0.924	0.809
NF1-Runtime	657.825	128.723	312.703	143.754	-	-	-	-
NF2-Avg. Delay	0.065	0.008	1.144	0.255	0.570	0.612	0.537	0.473
NF2-Runtime	602.285	848.922	836.940	803.185	-	-	-	-
NF3-Avg. Delay	0.045	0.025	0.149	0.009	0.885	0.479	0.703	0.646
NF3-Runtime	623.797	1014.218	826.203	1047.968	-	-	-	-

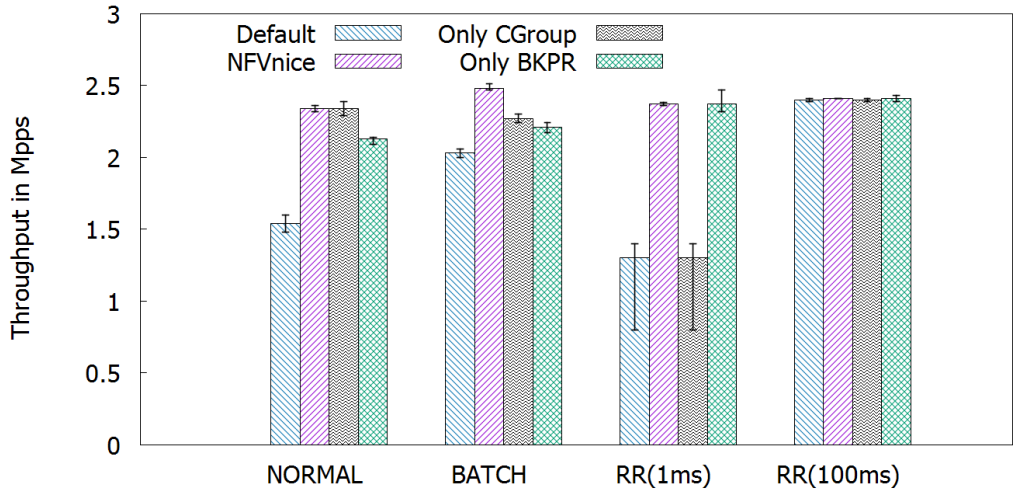


Figure 5.6: Performance of NFVnice in a service chain of 3 NFs with different computation costs

upstream NFs in the service chain.

5.4.3.1 *NF Service Chain on a Single Core:*

Here, we first consider a service chain of three NFs; with computation cost Low (NF1, 120 cycles), Medium (NF2, 270 cycles), and High (NF3, 550 cycles). All NFs run on a single shared core.

Figure 5.6 shows that NFVnice achieves an improvement of as much as a factor of two times in throughput (especially over the RR scheduler). We separately show the contribution of the `cgroups` and `backpressure` features. By combining these, NFVnice improves the overall throughput across all three kernel scheduling disciplines. Table 5.1 shows the number of packets dropped at either of the upstream NFs, NF1 or NF2, after processing (an indication of truly wasted work). With-

Table 5.3: Throughput, CPU utilization and wasted work in chain of 3 NFs on different cores

	Default			NFVnice		
	Svc. rate	Drop rate	CPU Util	Svc. rate	Drop rate	CPU Util
NF1 (∼550cycles)	5.95Mpps	-	100%	0.82Mpps	-	11% ±3%
NF2 (∼2200cycles)	1.18Mpps	4.76Mpps	100%	0.72Mpps	150Kpps	64% ±1%
NF3 (∼4500cycles)	0.6Mpps	0.58Mpps	100%	0.6Mpps	70Kpps	100%
Aggregate	0.6Mpps	-	300%	0.6Mpps	-	175% ±3%

out NFVnice, the default schedulers drop millions of packets per second. But with NFVnice, the packet drop rate is dramatically lower (near zero), an indication of effective avoidance of wasted work and proper CPU allocation.

We also gather perf-scheduler statistics for the average scheduling delay and run-time of each of the NFs. From Table 5.2, we can see that i) with NFVnice the run-time for each NF is apportioned in a cost-proportional manner (NF1 being least and NF3 being most), unlike the NORMAL scheduler that seeks to provide equal allocations independent of the packet processing costs. ii) the average scheduling delay with NFVnice for the NFs (that is the time taken to begin execution once the NF is ready) is lower for the NFs with higher processing time (which is exactly what is desired, to avoid making a complex NF wait to process packets, and thus avoiding unnecessary packet loss). Again this is better than the behaviour of the default NORMAL or RR schedulers [22] .

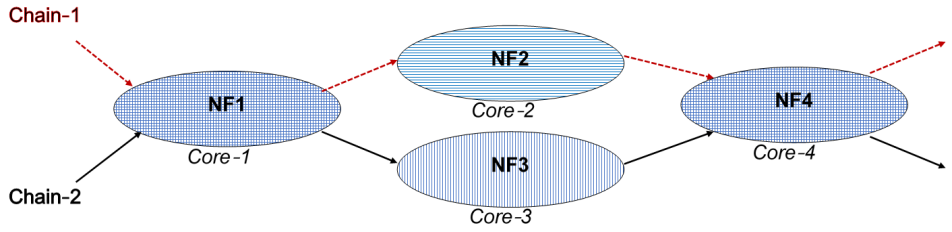


Figure 5.7: Different NF chains (Chain-1 and Chain-2, of length three), using shared instances for NF1 and NF4.

[22]Even though, for this experiment, RR(100ms) performs as well as NFVnice, it performs very poorly with variable per-packet processing costs, as seen in 5.4.4.1 and for chains with heterogeneous computation costs, as in 5.4.4.2 scenarios.

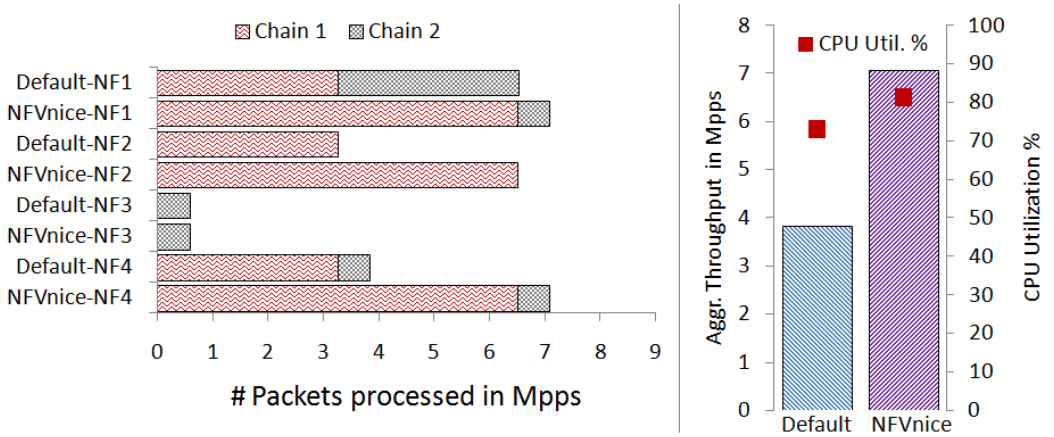


Figure 5.8: Multi-core chains: Performance of NFVnice for two different service chains of 3 NFs (each NF pinned to a different core), as shown in Fig. 5.7.

Table 5.4: Throughput, CPU utilization and wasted work in a chain of 3 NFs (each NF pinned to a different core) with different NF computation costs

		Default			NFVNice		
		Svc.Rate (pps)	Drop Rate (pps)	CPU Util.%	Svc.Rate (pps)	Drop Rate (pps)	CPU Util.%
NF1 (∼270cycles)	Chain1	3.26M	2.86M	78.6% ±0.4	6.498M	0	82.1% ±0.5
	Chain2	3.26M			0.583M		
	Aggregate	6.522M			7.08M		
NF2 (∼120cycles)	Chain1	3.26M	∼0	52.8% ±1.2	6.498M	∼0	58% ±0.7
	Chain2	-			-		
	Aggregate	3.26M			6.498M		
NF3 (∼4500cycles)	Chain1	-	2.68M	100% ±0	-	<100	100% ±0
	Chain2	0.582M			0.582M		
	Aggregate	0.582M			0.582M		
NF4 (∼300cycles)	Chain1	3.26M	0	60% ±0.7	6.498M	0	84% ±0.7
	Chain2	0.582M			0.582M		
	Aggregate	3.842M			7.08M		

5.4.3.2 Multi-core Scalability

We next demonstrate the benefit of NFVnice with the NFs in a chain across cores, with an NF being pinned to a separate, dedicated core for that NF. We use these experiments to demonstrate the benefits of NFVnice, namely: a) avoiding wasted work through backpressure; and b) judicious resource (CPU cycles) utilization through scheduling. When NFs are pinned to separate cores, there is no specific role/contribution for the vanilla OS schedulers, and for such an experiment we use the default

scheduler (NORMAL).

First, we consider the chain of 3 NFs, NF1 (Low, 550 cycles), NF2 (Medium, 2200 cycles) and NF3 (High, 4500 CPU cycles). Compared to the default scheduler (NORMAL), NFVnice plays a key role in avoiding the wasted work and efficiently utilizing CPU cycles. Table 5.3 shows that NFVnice's CPU utilization by NF1 and NF2 on their cores is dramatically reduced, going down from 100% to $\tilde{1}1\%$ and 64% respectively, while maintaining the aggregate throughput (0.6 Mpps). This is primarily because of backpressure ensuring that the upstream NFs only process the correct amount of packets that the downstream NFs can consume. Excess packets coming into the chain are dropped at the beginning of the chain. When we use only the default NORMAL scheduler by itself, NF1 and NF2 use 100% of the CPU to process a huge number of packets (the 'service rate' in the Table 5.3), only to be discarded at the downstream NF3.

We now consider two different service chains using 4 cores in the system. Chain-1 has three NFs: NF1 (270 cycles), NF2 (120 cycles) and NF4 (300 cycles) running on 3 different cores. Chain-2 comprises NF1, NF3(4500 cycles) and NF4. The same instances of NF1 and NF4 are part of both chain-1 and chain-2 as shown in Figure 5.7. Moongen generates 64-byte packets at line rate, equally splitting them between two flows that are assigned to chain-1 and chain-2. Table 5.4 shows that in the Default case (NORMAL scheduler), NF1 processes almost an equal number of packets for chain-1 and chain-2. However, for chain-2, the downstream NF3 discards a majority of the packets processed by NF1. This results not only in wasted work, but it also adversely impacts the throughput of chain-1. On the other hand, with NFVnice, backpressure has the upstream NF1 process only the appropriate number of packets of chain-2 (which has its bottleneck at the downstream NF, NF3). This frees up the upstream NF1 to use the remaining processing cycles to process packets from chain-1. NFVnice improves the throughput of chain-1 by factor of 2. At the same time, it maintains the throughput of chain-2 at its bottleneck (NF3) rate of 0.6Mpps. Overall, NFVnice not only avoids wasted work, but judiciously allocates CPU resources (at upstream NFs) proportionate to the chain's bottleneck resource capacity as shown in the Figure 5.8.

5.4.4 Salient Features of NFVnice

5.4.4.1 Variable NF packet processing cost

We now evaluate the resilience of NFVnice to not only heterogeneity across NFs, but also variable packet processing costs within an NF. We use the same three-NF service chain used in 5.4.3.1, but modify their processing costs. Packets of the same

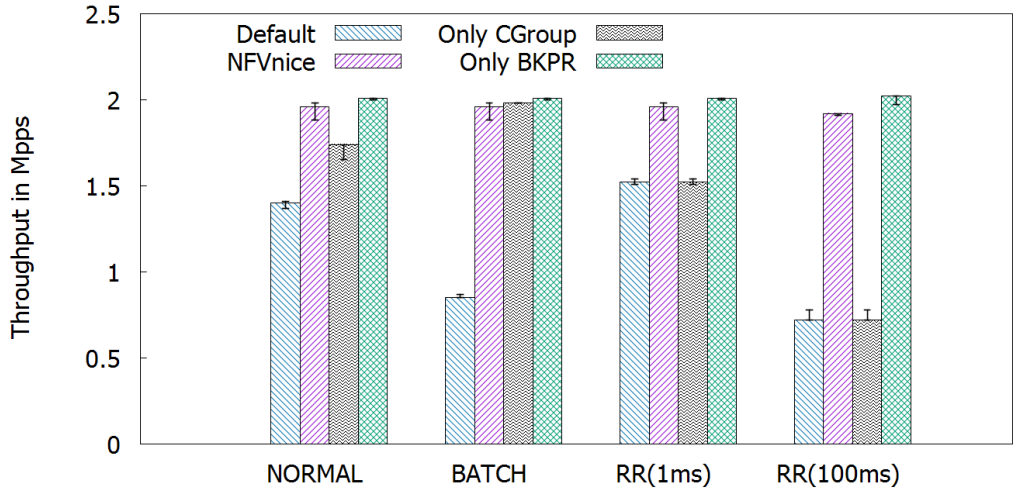


Figure 5.9: Performance of NFVnice in a service chain of 3 NFs with different computation costs and varying per packet processing costs.

flow have varying processing costs of 120, 270 or 550 cycles at each of the NFs. Packets are classified as having one of these 3 processing costs at each of the NFs, thus yielding 9 different variants for the total processing cost of a packet across the 3 -NF service chain. Figure 5.9 shows the throughput for different schedulers. With the Default scheduler, the throughput achieved differs considerably compared to the case with fixed per-packet processing costs as seen in Figure 5.6. For the Default scheduler, the throughput degrades considerably for the vanilla coarse time-slice schedulers (BATCH and RR(100ms)), while the NORMAL and RR(1ms) schedulers achieve relatively higher throughputs. When examining the throughput with only the CPU weight assignment, `CGroup`, we see improvement with the BATCH scheduler, but not as much with the NORMAL scheduler. This is because the variation in per-packet processing cost of NFs result in an inaccurate estimate of the NF's packet-processing cost and thus an inappropriate weight assignment and CPU share allocation. This inaccuracy also causes NFVnice (which combines `CGroup` and backpressure) to experience a marginal degradation in throughput for the different schedulers. Backpressure alone (the Only BKPR case), which does not adjust the CPU shares based on this inaccurate estimate is more resilient to the packet-processing cost variation and achieves the best (and almost the same) throughput across all the schedulers. NFVnice gains this benefit of backpressure, and therefore, in all cases NFVnice's throughput is superior to the vanilla schedulers. We could mitigate the impact of variable packet processing costs by profiling NFs more precisely and frequently, and averaging the processing over a larger window of packets. However, we realize that this can be expensive, consuming considerable CPU cycles

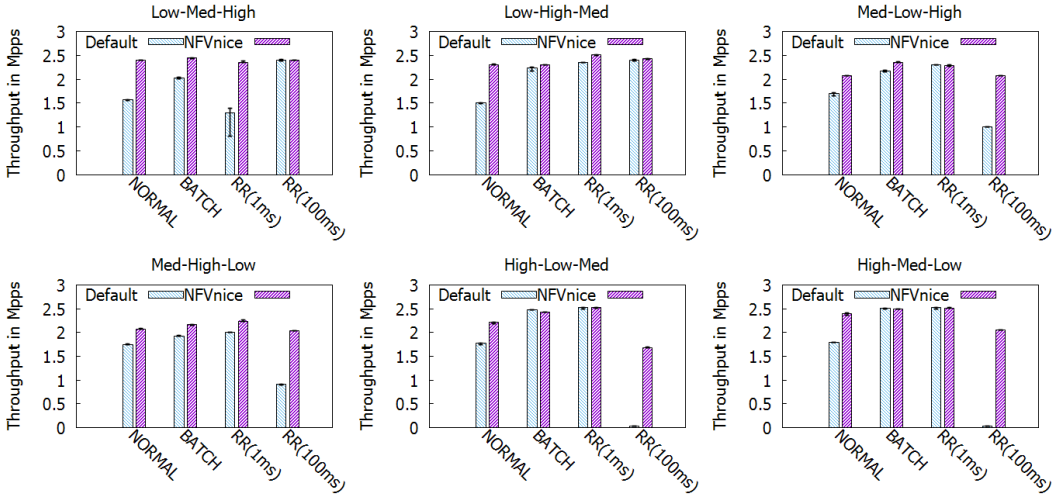


Figure 5.10: Throughput for varying combinations of 3 NF service chain with Heterogeneous computation costs

itself. This is where NFVnice's use of backpressure helps overcome the penalty from the variability, getting better throughput and reduced packet loss compared to the default schedulers.

5.4.4.2 Service Chain Heterogeneity

We next consider a three NF chain, but vary the chain configuration—(Low, Medium, High);(High, Medium, Low); and so on for a total 6 cases—so that the location of the bottleneck NF in the chain changes in each case. Results in Figure 5.10 show significant variance in the behaviour of the vanilla kernel schedulers. NORMAL and BATCH perform similar to each other in most cases, except for the small differences for the reasons described earlier in Section 3.2. We also looked at RR with time slices of 1ms and 100ms, and their performance is vastly different. For the small time-slice, performance is better when the bottleneck NF is upstream, while RR with a larger time-slice performs better when the bottleneck NF is downstream. This is primarily due to wasted work and inefficient CPU allotment to the contending NFs. However, with NFVnice, in almost every case, we can see considerable improvements in throughput, for all the schedulers. NFVnice minimizes the wasted cycles independent of the OS scheduler's operational time-slice.

Impact of RR's Time Slices with NFV: Consider the chain configurations "High-Med-Low" and "Med-High-Low" in Figure 5.10. RR(100 ms time slice) performs very poorly, with very low throughput $< 40Kpps$. This is due to the 'Fast-

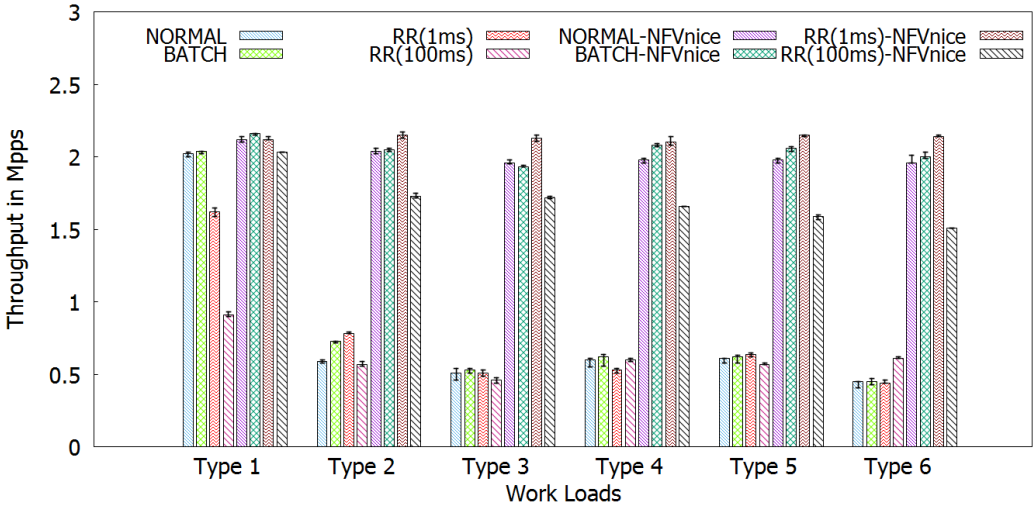


Figure 5.11: Throughput (Mpps) with varying workload mix, random initial NF for each flow in a 3 NF service chain (homogeneous computation costs)

producer, slow-consumer' situation [100], making the NF with "High" computes hog the CPU resource. Now, in the default RR scheduler, the packets processed by this NF would be dequeued by the Tx threads but will be subsequently dropped, as the next NF in the chain does not get an adequate share of the CPU to process these packets. The upstream NF that is hogging the CPU has to finish its time slice and the OS scheduler then causes a involuntary context switch for this "High" NF. However, with NFVnice, the queue buildup results in generating a backpressure signal across the chain, forcing the upstream NF to be evicted (i.e., triggering a voluntary context switch) from the CPU as soon as the downstream NFs buffer levels exceed the high watermark threshold. The upstream NF will not execute till the downstream NF gets to consume and process its receive buffers. Thus, NFVnice is able to enforce judicious access to the CPU among the competing NFs of a service chain. We see in every case in Figure 5.10, NFVnice's throughput is superior to the vanilla scheduler, emphasizing the point we make in this paper: NFVnice's design can support a number of different kernel schedulers, effectively support heterogeneous service chains and still provide superior performance (throughput, packet loss).

5.4.4.3 Workload Heterogeneity

We next use 3 homogeneous NF's with the same compute cost, but vary the nature of the incoming packet flows so that the three NFs are traversed in a different order for each flow. We increase the number of flows (each with equal rate) arriving

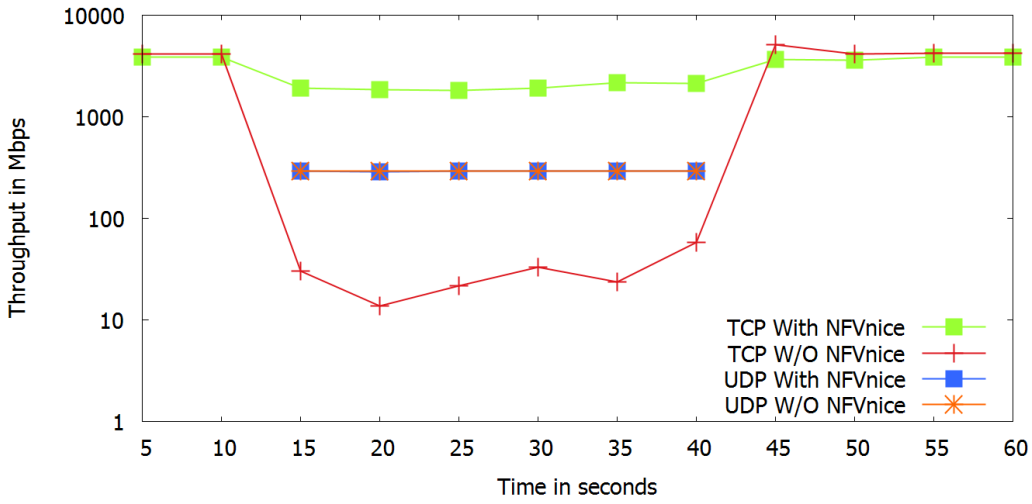


Figure 5.12: Benefit of Backpressure with mix of responsive and non-responsive flows, 3 NF chain, heterogeneous computation costs

from 1 to 6, as we go from Type 1 to Type 6, with each flow going through all 3 NFs in a random order. Thus, the bottleneck for each flow is different. Figure 5.11, shows that the native schedulers (first four bars) perform poorly, with degraded throughput as soon as we go to two or more flows, because of the different bottleneck NFs. However, NFVnice performs uniformly better in every case, and is almost independent of where the bottlenecks are for the multiple flows. Moreover, NFVnice provides a substantial improvement and robustness to varying loads and bottlenecks even across all the schedulers (NORMAL, BATCH, RR with 1ms or 100 ms slice.)

5.4.4.4 Performance isolation

It is common to observe that when there are responsive (TCP) flows that share resources with non-responsive (UDP) flows, there can be a substantial degradation of TCP performance, as the congestion avoidance algorithms are triggered causing it to back-off. This impact is exacerbated in a software-based environment because resources are wasted by the non-responsive UDP flows that see a downstream bottleneck, resulting in packets being dropped at that downstream NF. These wasted resources result in less capacity being available for TCP. Because of the per-flow backpressure in NFVnice, we are able to substantially correct this undesirable situation and protect TCP's throughput even in the presence of non-responsive UDP.

In this experiment, we generate TCP and UDP flows with Iperf3. One TCP flow goes through only NF1 (Low cost) and NF2 (Medium cost) on a shared core. 10 UDP

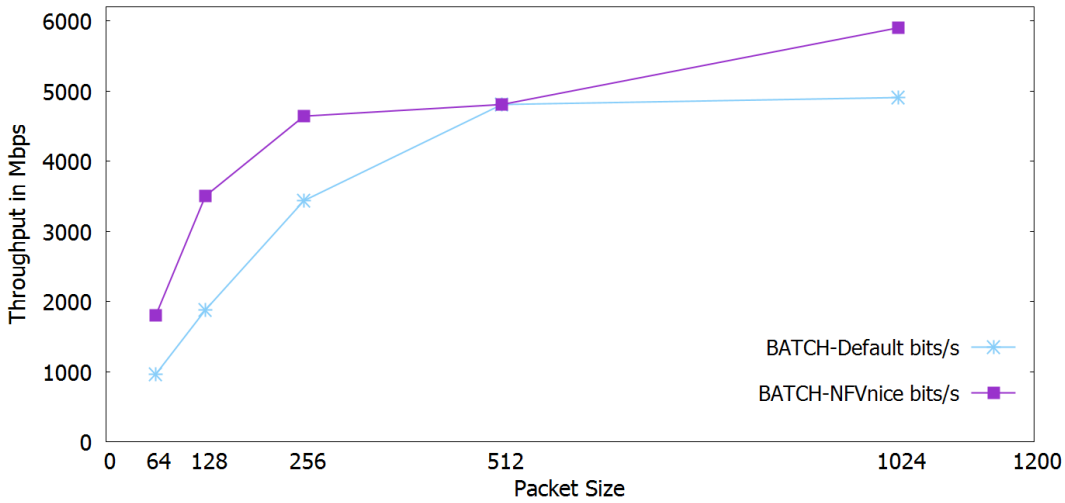


Figure 5.13: Improvement in Throughput with NFs performing Asynchronous I/O writes withNFVnice

flows share NF1 and NF2 with the TCP flow, but also go through an additional NF3 (High cost, on a separate core) which is the bottleneck for the UDP flows - limiting their total rate to 280 Mbps.

We first start the 1 TCP flow. After 15 seconds, 10 UDP flows start, but stop at 40 seconds. As soon as the UDP flows interfere with the TCP flow, there is substantial packet loss without NFVnice, because NF1 and NF2 see contention from a large amount of UDP packets arriving into the system, getting processed and being thrown away at the queue for NF3. The throughput for the TCP flow craters from nearly 4 Gbps to just around 10-30 Mbps (note log scale), while the total UDP rate essentially keeps at the bottleneck NF3's capacity of 280 Mbps. With NFVnice, benefiting from per-flow backpressure, the TCP flow sees much less impact (dropping from 4 Gbps to about 3.3 Gbps), adjusting to utilize the remaining capacity at NF1 and NF2. This is primarily due to NFVnice's ability to perform selective early discard of the UDP packets because of the backpressure. Otherwise we would have wasted CPU cycles at NF1 and NF2, depriving the TCP flow of the CPU. Note that the UDP flows' rate is maintained at the bottleneck rate of 280 Mbps as shown in Figure 5.12 (UDP lines are one on top of the other). Thus, NFVnice ensures that non-responsive flows (UDP) do not unnecessarily steal the CPU resources from other responsive (TCP) flows in an NFV environment.

5.4.4.5 Efficient I/O handling by NFVnice

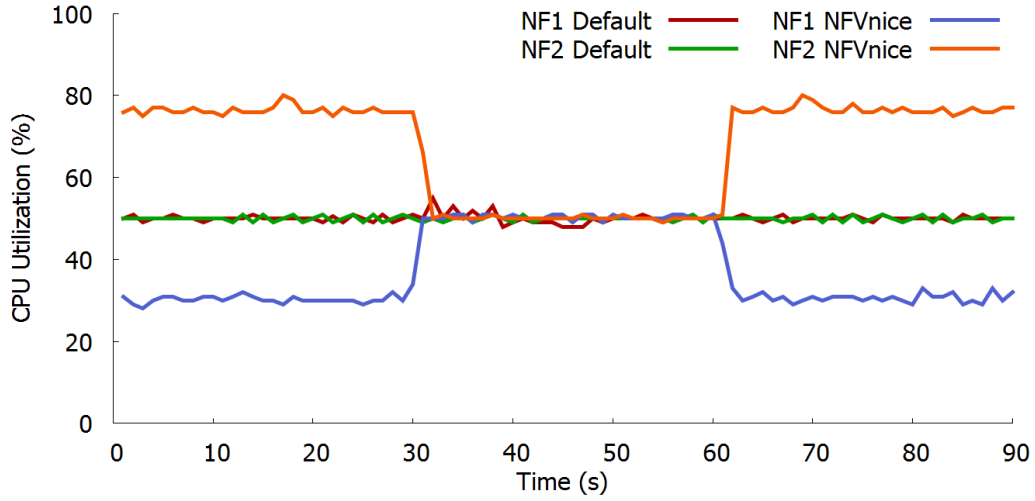
It is important for NFs to be able to perform I/O required by the packet of a flow, while efficiently continuing to process other flows (e.g., packet monitors, proxies, etc.). Using Moongen we send 2 flows at line rate. Both the flows share the same upstream NFs, but only one of the flows performs I/O *i.e.,* logs the packets to the disk using NFVnice's I/O library. Figure 5.13 compares the aggregate throughput achieved with and without NFVnice, using the BATCH scheduler in the kernel. We vary the packet size. NFVnice maintains a higher throughput consistently, even for small packet sizes. Moreover, NFVnice maintains progress on the second flow while I/O is being performed for packets of the first flow, thus providing better isolation.

5.4.4.6 Dynamic CPU Tuning and fairness

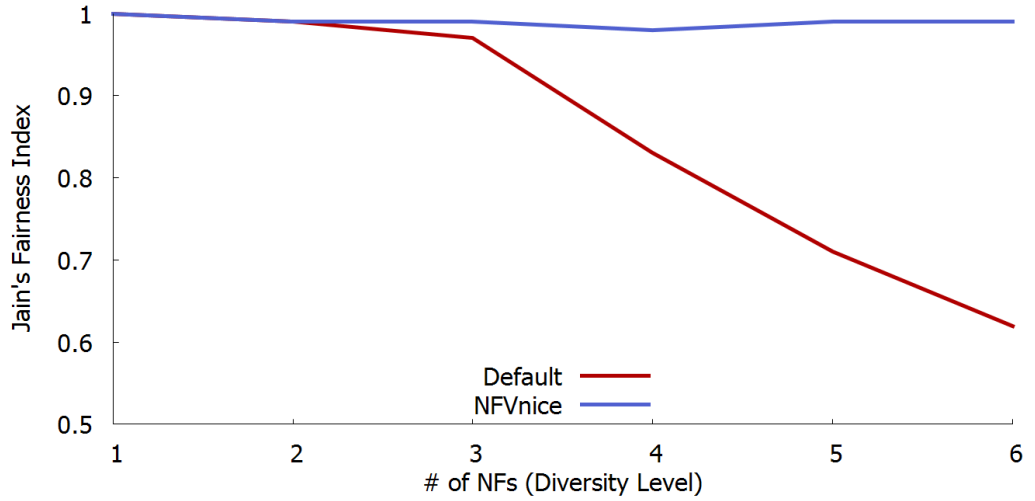
Dynamic CPU tuning: NFVnice dynamically adjusts the CPU allocations based on the packet processing cost and arrival rate for each NF. Two NFs initially with different computation costs (ratio 1:3) run on the same core, with MoonGen transmitting a flow each to the two NFs at the same rate. To demonstrate adaptation, we have the computation cost of NF1 temporarily increase 3 times(to the same level as NF2) during the 31 sec. to 60 sec. interval.

Figure 5.14a has the default NORMAL scheduler evenly allocating the CPU between NF1 and NF2 regardless of their computation cost throughout. On the other hand, NFVnice allocates NF2 three times the CPU as NF1 initially. At t=30s, NFVnice allocates each NF half of the CPU. And at t=60s, we go back to the original allocation. We observed that the throughput for the two flows (not shown) is equal throughout, indicating the capability of NFVnice to dynamically provide a fair allocation of resources factoring in the heterogeneity of the NF CPU compute cost.

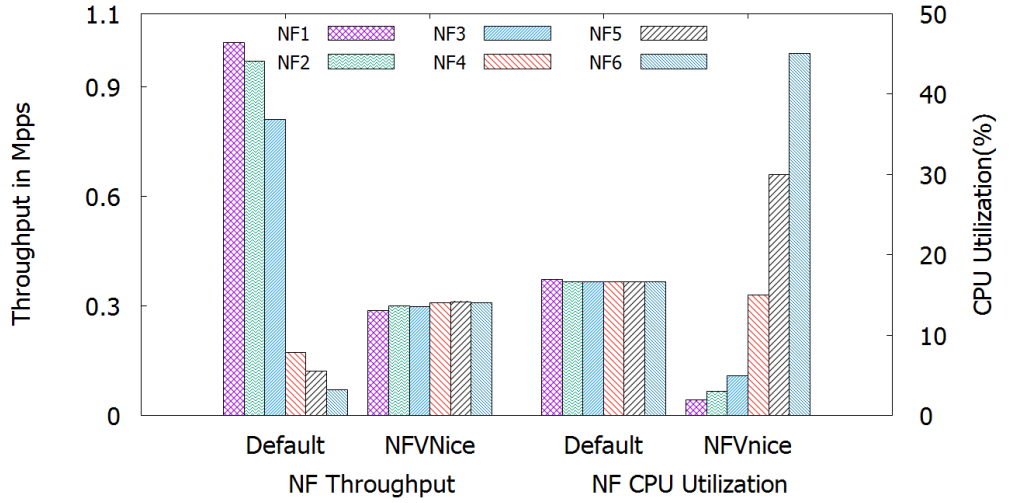
Fairness measure: We evaluate the fairness in throughput as we increase the diversity of computation for each of the NFs for default CFS scheduler and NFVnice. We vary the number of NFs sharing the core. Each NF has the same packet arrival rate, but different computation cost. At diversity level 1, we start with a single flow (uses NF1, compute cost 1). With a diversity level of two, we have 2 flows, flow 1 uses NF1 (compute cost 1), flow 2 uses NF2 (compute cost 2). At a diversity level of 6, there are 6 NFs, with the ratio of computation costs of 1:2:5:20:40:60, and one flow each going to the corresponding NF. At diversity level 6, the NORMAL scheduler allocates 16.6% of the CPU to each of the NFs, being unaware of the computation cost of each NF. Thus, the throughput for flow 1 is 1.02 Mpps, while flow 6 is only 0.07 Mpps. With NFVnice, the CPU allocated to the lightweight NF is 1%, while the heavyweight NF gets 46%, and all the flows achieve nearly equal throughput 5.14c).



(a) Effect of Dynamic CPU Weight Updates



(b) Measure of Fairness



(c) Effect of rate-cost proportional fairness on CPU Utilization and Throughput

Figure 5.14: Adaptation to Dynamic Load and Fairness measure of NFVnice compared with the NORMAL scheduler

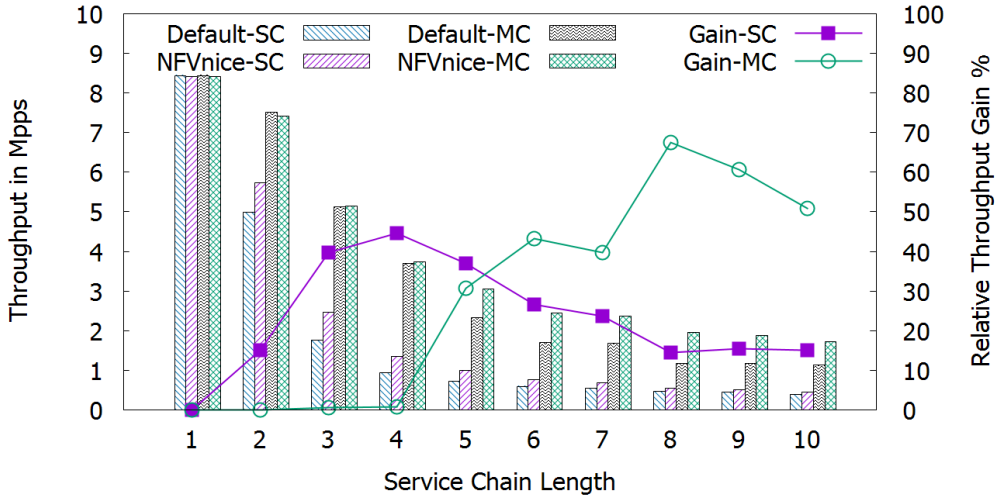


Figure 5.15: Performance of NFVnice for different NF service chain lengths.

Using Jain's fairness index [101], we show that the vanilla scheduler is dramatically unfair (going down to 0.62) while NFVnice consistently achieves fair throughout (Jain's fairness index of 1.0) as shown in figure 5.14b).

5.4.4.7 Supporting longer NF chains

We now see how well NFVnice can support longer NF service chains. We choose three different NFs, as in 5.4.3, and increase the chain length from 1 NF up to a chain of 10 NFs, including one of the 3 NFs each time. We examine two cases: (i) all the NFs of the chain are on a single core (denoted by SC); and (ii) three cores are used, and as the chain length is increased, the additional NF is placed on the next core in round-robin fashion (represented by MC). Results are shown in Figure 5.15. For the single core, NFVnice achieves higher throughput than the Default scheduler for longer chains, with the greater improvements achieved for chain lengths of 3-6. As the chains get longer (>7 NFs sharing the same core), the improvement with NFVnice is not as high. For the multiple core case, NFVnice improves throughput substantially, especially as more NFs are multiplexed on a care (e.g., chain lengths > 4), compared to the Default scheduler. Of course, the improvement with NFVnice will depend on the type of NFs and their computation costs, for individual use-cases.

5.5 Conclusion

As the use of highly efficient user-space network I/O frameworks such as DPDK becomes more prevalent, there is going to be a growing need to mediate application-level performance requirements across the user-kernel boundary. OS-based schedulers lack the information needed to provide higher level goals for packet processing, such as rate proportional fairness that needs to account for both NF processing cost and arrival rate. By carefully tuning scheduler weights and applying backpressure to efficiently shed load early in the the NFV service chain, NFVnice provides substantial improvements in throughput and drop rate and dramatically reduces wasted work. This allows the NFV platform to gracefully handle overload scenarios while maintaining efficiency and fairness.

Our implementation of NFVnice demonstrates how an NFV framework can efficiently tune the OS scheduler and harmoniously integrate backpressure to meet its performance goals. Our results show that selective backpressure leads to more efficient allocation of resources for NF service chains within or across cores, and scheduler weights can be used to provide rate proportional fairness, regardless of the scheduler being used.

Chapter 6

Future Prospects

In this chapter we consider and present the future prospects of userspace NF scheduling mechanisms proposed in NFVnice, in the view of alternative NFV platforms and technology advancements.

Contents

6.1 Applicability of NFVnice in other NFV Platforms

6.1.1 ClickOS

Scheduling of click elements in ClickOS relies on MiniOS, which has a single address space, and runs a co-operative scheduler [47]. Thus network elements need to voluntarily relinquish CPU to allow the other runnable network elements to be scheduled. Also, the packet processing is performed in burst mode, where a batch of packet are

processed at a time by distinct network functions. Backpressure mechanism readily fits in this model and can significantly benefit in reducing the wasted-work and optimizing the packet processing across NF chains.

6.1.2 NetBricks

NetBricks employs DPDK run-to-completion model where a packet upon arrival is processed by chain of NFs, (*i.e.,* all NFs process the packet atomically within a single processing context) before transmitting out the packet. As there is no distinction between processing by an NF or distinct NFs within a chain, cgroups or backpressure mechanisms cannot be directly applied. However, when multiple such NF chains are deployed, cgroups can be leveraged to provide efficient CPU allocation to all the contending NF chains, to ensure fair and proportional allocation of CPU.

6.2 Current Limitations and Prospects of Extensions

Here, we list the current limitations of NFVnice and briefly describe the means to improve and overcome these shortcomings.

6.2.1 Cross-Node Backpressure

The backpressure signaling is restricted to chains withing a single NFV node managed by single VNFM. When the NF chain spans multiple nodes, each VNFM has visibility only for the subset of NFs that run on that particular node, hence can apply and enforce backpressure only for the part of the chain that is deployed on each node. However, enabling cross-node backpressure is non-trivial for the following reasons:

- To ensure backpressure across nodes, the prerequisite is to extend the cross-node chain-wide visibility to each of the involved VNFMs.
- Next, we need some explicit notification mechanism (out-of-band) to distinctly distinguish and identify the NF chains across all the VNFMs.
- In addition, the transmission delay (that is usually higher order than local packet processing) also needs to be accounted and profiled for distinct chains, without which the system can result in significant performance degradation.

Nonetheless, the system can be enhanced to dynamically track per-chain, per-node communication latency and by utilizing simple/customized notification mechanisms like *Internet Control Message Protocol* (ICMP) source quench across VNFM cross-

node backpressure can be realized.

6.2.2 Accounting Delay Constraints

NFVnice provides a scalable, high performance NFV platform by allowing to efficiently multiplex and schedule the NFs. However, the consequence of multiplexing multiple NFs on a same core is the increased per-packet processing delay. While the overall processing delay from NF perspective is bound by the scheduling sequence and quantum of scheduled time for each of the contending NF; these parameters alone are not sufficient to determine the end-to-end delay experienced by per-packet. This measurement requires chain-aware delay tracking either at the per-packet level or at the aggregate per-flow level. Currently, there is no mechanism to track the delay.

NFVnice can be easily enhanced to measure the packet processing delay at each sequence of NFs in the chain and accordingly adapt scheduling decisions and apply the polices to better suit the delay constraints of distinct flows. For example, in the simple case, the flows that have exceed the delay constraints can be notified to VNFM so that the a) flows can be signaled to backpressure in order to inhibit any further processing of the packets of those flows to ensure early drop without wasting of any extra CPU resources; or b) to re-route flows through alternative less-congested NFIs. Also, as discussed earlier, enhancement to account delay for chains spanning multiple nodes can also be achieved, provided the cross-node communication protocol is setup.

6.3 Prospects of NFVnice with other advancements

6.3.1 Micro services

Microservices is a software development/architectural pattern that structures an application as a collection of loosely coupled services [102]. Microservices enable modular, distributed software components that can be independently scaled and deployed to provide a service. This architecture has garnered lot of attention in cloud based services, especially in the Telecommunications; the likes of AT&T, BT, CenturyLink, and Telefonica have publicly embraced the move to a microservices architecture in the Telecommunications cloud[23] [103, 104].

However, Microservices account to added latency and performance degradation due to increased modular communication and state transfers [105]. In addition,

[23] AT&T highlighted on the role of microservices in their goal to virtualize 75% of their network by 2020; BT announced that containers, a form of microservices, will be used to build their NFVI.

Microservices readily yield to large chain of modular services to realize different network services. Hence, addressing chain-wide performance is pivotal. We can employ NFVnice (selective backpressure and cgroup weight settings) to account for chain-wide characteristics and also to efficiently multiplex and schedule these Microservices within a node, and avoid any wasted work. The effect will be more pronounced in containing the wasted work across the components of micro services. Also, the scheduler weights (cgroup) can be used to provide rate proportional fairness, without worrying about the choice of the underlying kernel scheduler used in scheduling the microservices. Further, to better suit the Microservices architecture, the cross-node selective backpressure signaling and ECN for end-to-end rate control are necessary.

6.3.2 UniKernels

In stark contrast to Microservices, Unikernels are specialised, single-address-space machine images that bundle just the necessary library operating systems. Unikernels offer more lean images, that are quick to boot, more secure and optimized for specific services. This approach along the lines of containers is gaining more Industry attention, especially in the NFV [106].

Owing to lean image and quick boot, Unikernels readily benefit on-demand dynamic provisioning and large scale deployment scenarios. However, the downside is the burden it places on the cloud orchestration layers because of the need to schedule many more instances with greater churn [107]. Though, NFVnice does not readily fit for this architecture, the rate-cost proportional scheduling principle of NFVnice can be leveraged in hypervisors (on which the unikernel instances run) to account for fair and judicious CPU allocation to all the active instances.

6.3.3 Enhanced Disk I/O Management

Storage Performance Development Kit (SPDK)[24] provides extremely high performance I/O processing (millions of I/Os per second) at user level by using the *poll-mode drivers*. Several NFs require frequent disk access to read/write/update packet information for flows *e.g.,* packet-loggers that typically log all the packet headers and sometimes the entire packets of specific flows to disk, caching proxy and file servers retrieve and update cached (in-memory) content and files from disks. Hence, providing an integrated NFV framework that utilizes SPDK alongside DPDK to provide more efficient framework for NFs is more desirable.

[24]For more details refer: https://software.intel.com/en-us/articles/introduction-to-the-storage-performance-development-kit-spdk.

Part II

Addressing Network-level Challenges in NFV Resource Management: Placement, Steering, and Load-balancing

Chapter 7

Problem Statement

Contents

7.1 Introduction

*Network Function*s (NFs) are becoming ubiquitous in today's networks. NFs or traditionally the Middleboxes (MBs) or the *Physical Network Function*s (PNFs) have typically been constructed from purpose-built hardware, customized to perform specific tasks. In traditional networks, once the network of MBs is setup, it cannot alter its structure (*e.g.,* topology is hardwired) or functionality (*e.g.,* fixed set of services, cannot morph from one service to another).*Network Function Virtualization* (NFV) has evolved the MB architectures to virtual or software-based services on top of commercial off-the-shelf (COTS) hardware, it provides the agility and increased flexibility in the network [108].

7.2 Problem Description

On one hand, NFV promises to increase flexibility and achieve efficiency in using network resources, since both the structure and the functionality of NFV nodes can be adjusted dynamically in response to service demand. But, on the other, it poses

several orchestration, management problems and deployment concerns (outlined in earlier section §1.1.1), especially in determining when and where to place the NFs, how to steer the traffic through these NFs, how to distribute the load across multiple instances of NFs to improve network resource utilization efficiency, QoS, *etc.* Also, *Service Function Chaining* (SFC), which determines the exact sequence of NFV-based middlebox services a flow has to pass through, is gaining momentum as a necessary network process in *Communication Service Provider* (CSP) networks must be accounted.

In order to exploit the full potential of NFV and to facilitate SFC, we posit the need for the following:

i) VNFIs have to be dynamically placed, replicated, instantiated and terminated or consolidated,

ii) new incoming flows have to be dynamically steered to the least-expensive NFI (*i.e.,* in terms of current network and computation load), and

iii) active, existing flows have to be redirected (if and when needed) to other instances of the same service in order to balance the load between the NFIs of this service.

iv) flexible and efficient traffic steering mechanism that does not overburden the switch TCAMs and enable to provide a topology independent *Service Function Chaining*.

7.2.1 Need for NFV Resource Management and Orchestration Framework

In order to address the aforementioned requirements, we introduce and make the case for *a resource management and orchestration framework that dynamically handles NFIs and flow traffic*, in order efficiently instantiate and place the VNFIs, steer the traffic through the active VNFIs and to load-balance i) the network load in links, and computation load in NFIs. In designing such a resource management framework, we consider all potential options, that is, from centralized, software-based control plane approaches to decentralized, hardware-oriented data plane approaches.

Centralized or Distributed Framework? We find that a centralized controller can become the bottleneck when assigned with the task of making *real-time decisions* on service placement, instantiation, termination and flow steering/redirection. On the other hand, a purely distributed approach suffers from increased latency and

overhead in order to exchange information between decision-making nodes, a process that also raises stability issues.

We argue for the need of a hybrid solution, where the *(logically) centralised SDN controller* performs lightweight tasks, related to the NFV environment coordination and flow setting-up of state; while the *distributed network of NFIs* makes more frequent decisions, that impact flow latency. Despite the multiple SDN-NFV architectures proposed in recent years (*e.g.,* E2 [22], Stratos [109], Slick [110], SDNFV [111]), the problem of resource allocation and management in such environments has not received as much attention.

7.2.2 SFC Management and Flow Steering

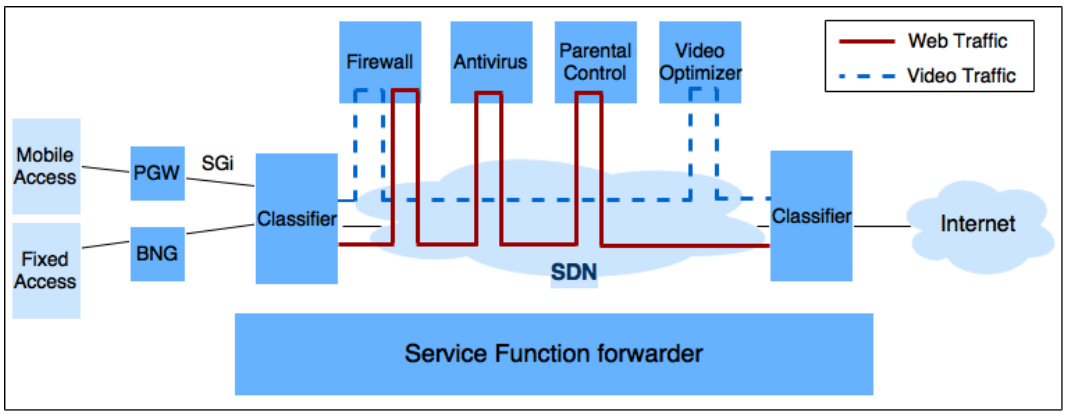


Figure 7.1: SFC Use case for two different traffic classes

Consider the Figure 7.1 that illustrates two high level policies *i.e.,* i) for the class of video traffic: the service function chain demands the traffic to traverse through Firewall and Video-optimizer service functions and ii) for the regular web traffic: Firewall and the value added services functions like anti-virus and parental control services are desired. In such cases, once the flows are classified, the flows need to be steered along two different paths to go through the respective policy desired specific service functions. SDN provides the flexibility in setting different flow rules (coarse or fine grained) across the switches and enables to steer flows accordingly. With NFV, the service functions can be provisioned to meet the traffic demands and thus many instances of network functions can be instantiated at will to ensure performance even at varying network loads.

Though, SDN and NFV provide richer support for more fine grained traffic steering and demand instantiation of network functions; there are still several challenges in realizing the dynamic SFC with traditional routing constructs.

Problem statement for SFC [16] lists out the key issues in realizing dynamic SFC with the traditional networks and illustrates the desired characteristics of dynamic SFC with elastic network functions. Herein, we present the key issues and characteristics most pertinent to realization of SFC.

a) Foremost is the topological and service function chain definition independence, that aim towards defining the SFC in such a way that mapping of policies to the desired service functions is independent of the topology and underlying network routing mechanism. This decoupling ensures the service policies to be more generic and easily deployable across different networks.

b) Second is the capability to re-classify and update the SFC, which enables to change the course of traffic to traverse through different service types. This feature is highly beneficial for providing additional security and value added services like intrusion detection, deep packet inspection, and URL filtering.

In the section 2.3, we presented *Network Service Header* (NSH) [33], which is targeted towards addressing the above concerns. We now outline and discuss on the NSH problem space and why there is a need to redress the proposed NSH.

7.2.3 Where NSH falls short?

Foremost, NSH is a SFC encapsulation, which is transport agnostic and requires an outer transport specific encapsulation to forward the NSH packet across the network. Control plane is responsible to manage this encapsulation. This however is customary function of a control plane even in the absence of NSH.

In addition, there are additional requirements on control plane to realize a NSH based SFC architecture. Some of the control plane requirements are partially described in SFC Architecture [41]. SFC control plane is responsible for constructing Service Function Paths (SFPs), translating SFCs to forwarding paths, and propagating path information to participating nodes to achieve requisite forwarding behavior to construct the service overlay. *i.e.,* It is up to the control plane to map the high level policies based on the network topology and service function instances in the network to specific Service Path Identifiers (management of SFPs) and in updating the SFFs about the SFP mapping, that can change over time with addition of new service function or deletion of existing services and instances.

Chapter 8

Related Work

In this Chapter, we present the literature survey on the state-of-the-art work in the prospect of a) load balancing the traffic across network links and NFs and b) flow steering approaches to realize SFC.

Contents

8.1 Network Load Balancing

Several works [8, 37, 112–118] have distinctly addressed the service function chaining, flow steering, NF placement, orchestration, and load balancing related aspects. Here we focus on and discuss some of the most-related state-of-the-arts works that specifically account for VNFI load balancing and SFC.

Figure 8.1 outlines the space of congestion control and load-balancing schemes proposed over past decade. We briefly discuss the most pertinent related works[25].

[25]The highlighted (blue) works in the Figure 8.1 are the only specific works in the NFV context, while the rest correspond to general traffic based congestion control and load-balancing.

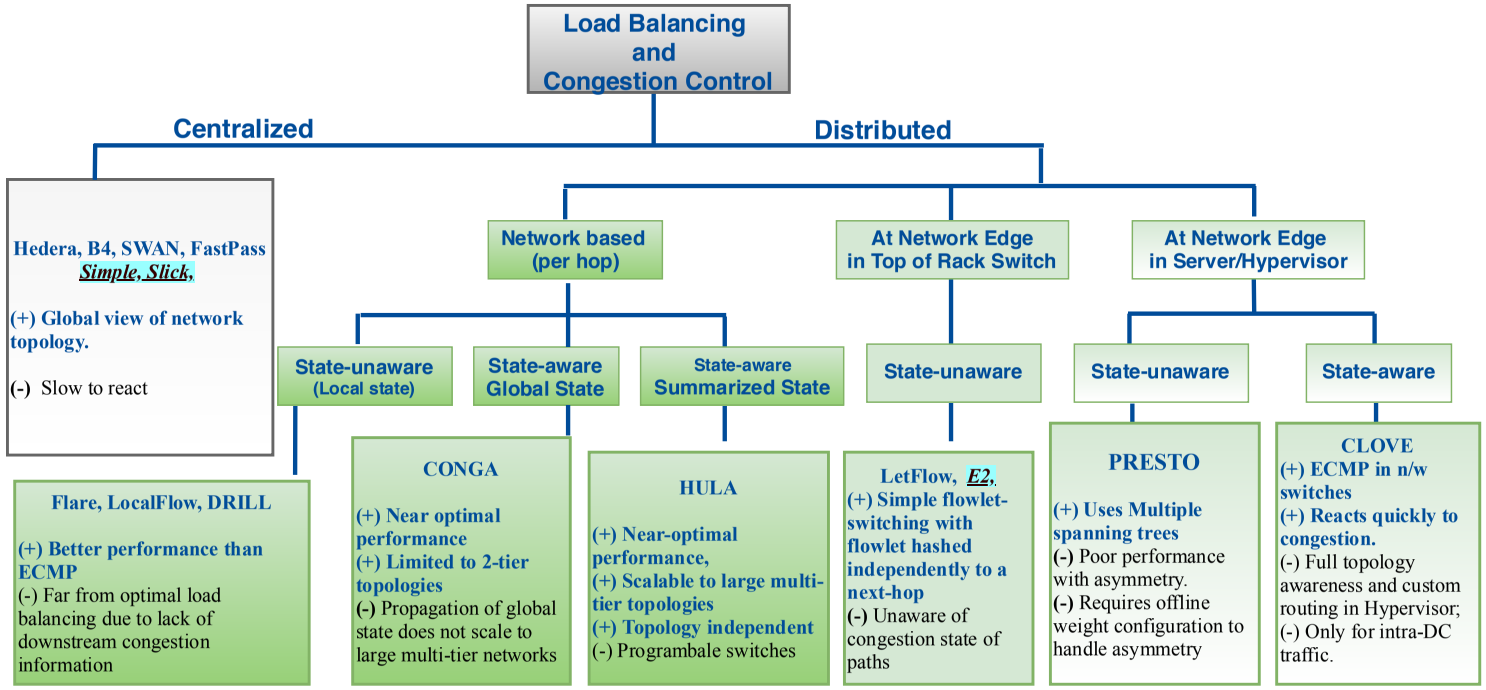


Figure 8.1: Classification and brief analysis of Congestion Control and Network Load Balancing literature.

8.1.1 Centralized Solutions

In the case of purely centralized solutions, the complexity involved in the decision making for service instantiation and flow redirection (NP-hard in many cases) results in a much coarser granularity of decision making. Hence, the solutions can only be based on heuristic-based approaches [117, 119], or act on a per-flow basis [120].

SIMPLE [23] primarily addresses the SDN based traffic steering approach that tries to optimize on the total forwarding rules at the switches. It relies on an offline Integer Linear Programming (ILP) solver to optimize the number of flow rules on the switches and an online Linear Programming (LP) solver for load balancing. However, SIMPLE [23] assumes that the NFIs and middleboxes are statically placed and any dynamic instantiation of NFIs requires the re-run of the expensive offline ILP solver.

Slick [110] provides a programming model abstraction for service chaining that supports heuristic based function placement and flow steering schemes. Slick [110] also supports dynamic scaling of NFIs. However, it does not redirect the existing flows from the overloaded instances, but only steers the new flows to the scaled-out instances.

By and large, related works in this area have targeted *only newly-arriving flows*, while works such as Split/Merge [72], and opennf [121] that address flow-redirection, are inefficient since they need to pause the existing flows for transferring VNFIs and forwarding states.

8.1.2 Distributed Solutions

On the other hand, purely distributed approaches for load balancing (*e.g.,* TeXCP [122], CONGA [123]) face a different set of issues, such as:

1) ***high overhead and latency:*** a large amount of information has to be exchanged among decision making nodes to synchronise their view of network state, with the consequent latency as the network scales;

2) ***complexity:*** achieving a stable synchronised view at every decision making node is complex; and

3) ***inefficiency and stability:*** since each node makes decisions based on its local view of the network state, packet loss, frequent rerouting, non-optimal load-balancing and transient loops are likely.

In [124], authors employ the concept of shadow-prices to trade-off performance, QoS,

and complexity, but they limit their scope only to the flow steering problem in SFC.

8.1.3 Network Function Load Balancing through Flow Redirection

OpenNF [121] presents a control plane for managing both network forwarding state and internal NFI state for migrating flows, where migration events are generated by NFIs and buffered at the controller for the interval of state migration.

Stratos [109] proposes an orchestration framework that employs a rack-aware NFIs placement strategy with horizontal scaling and migration of NFIs. The load/flow distributions are computed by an ILP formulation where only new flows are steered to the new instances while the existing ones continue to get processed at the same NFI, without alleviating the load on the already congested instances.

Lastly, E2 [22] is a NFV scheduling framework that supports affinity based placement and dynamic scaling of NFIs. The framework tries to minimize the traffic across switches while balancing the traffic across NFIs, but it avoids flow redirections across hardware switches. The VNFI instantiation decisions are taken locally and independently at each of the E2 data plane element and hence the scope of load-balancing is also local and restricted to load characteristics of individual VNFI and load-balancing decisions do not account for overall network load.

8.2 Flow Steering in Service Function Chains

Over the past few years, several works [22, 23, 25, 111, 118, 125–128] have proposed solutions for realizing SFC. We briefly discuss a few that employ network overlay (MPLS, VLAN, VxLAN), underlay (overloading the existing L2/L3/L4 header fields), and alternate header based approaches.

8.2.1 SFC with Network Overlay and Underlay

Shadow-MACs [125] and OpenSCaaS [126] emphasize on utilizing the L2 address fields —media access control (MAC) address to represent the path identifiers. [125] utilizes the destination MAC addresses as opaque forwarding labels while [126] employs the source MAC addresses as a forwarding label to setup the service chain ID (SC-ID). In both cases, the SDN controller has the responsibility of managing (defining and mapping) the SC-ID and setting up the appropriate L2 address to steer traffic to the desired service instance. StEERING [25] also overloads the L2 address fields, to steer packets to inline service functions. It relies on multiple forwarding tables that require additional extension to the Openflow API. In addition, it requires

all the service function instances in the network to be aware of the Ethernet address of all the service functions in the network topology, which limit the flexibility and scalability of elastic network functions.

These approaches provide efficient mechanism to steer the packets through the service function chain by re-purposing the existing packet headers fields, however they cannot support the exchange of meta-data and lack the ability to re-classify or alter the course of service functions after initial classification and assignment of the service function chain, thus inhibit the support for elastic network functions.

8.2.2 SFC with explicit tag and other alternatives

FlowTags [127] enable SFC by defining tag enhanced network functions, where the network functions generate and consume the service tags, while switches forward the packets based on the tags. SIMPLE [23] addresses efficient routing by constraining the number of switch forwarding rules and load balancing the traffic across middlebox instances and relies on the tag-based approach to tunnel packets across service functions. The computation and optimization of service paths is done through the mix of offline and online mixed integer linear programming, which results in considerable amount of computation time complexity. These approaches can facilitate re-classification and enable to alter the route through every service function, but as with earlier approaches, information exchange and sharing of meta-data is not possible. In addition these approaches need to account for additional complexity in tag management and distribution.

[129] and [37] propose *Information Centric Networking* (ICN) based approach of named services and named service instances for service chaining *i.e.,* the routing based on service function names and service instance names respectively. The network elements (routers and switches) take the responsibility of steering traffic to the desired service function instance. Both rely on network overlay/underlay mechanisms to tunnel flows to service instances. Though, the approach in [129] results in enormous reduction of switch rules, due to lack of fine grained control, it fails to provide visibility and control over appropriate service instance selection for a class of traffic, which is generally required for multi-tenancy, multi-subscriber policy matching and cannot cater to optimal instance utilization.

Chapter 9

Orchestration and Resource Management Framework: DRENCH

Contents

In this Chapter, we propose a semi-DistRibutEd orchestration and resource management framework for NFV based service function CHaining (**DRENCH**). To the best of our knowledge, *DRENCH is the first work that tackles both NFI placement and flow steering problems in arbitrary topologies.* DRENCH provides a computationally feasible algorithmic resource management framework inspired by the principles of market competition.

DRENCH incorporates a traffic load-balancing algorithm that utilises dynamic estimation of NFI loads with each NFV node independently directing flows to an appropriate least-loaded service instance; DRENCH utilises a real-time service instantiation capability and redirects existing flows as necessary; DRENCH is applicable for an NFI based Service Function Chaining environment by using a centralised SDN controller to disseminate information among the NFV nodes in the network. DRENCH as a resource management framework can fit into most of the existing architectures, albeit with some modifications.

We realize DRENCH in the context of SDN-NFV architectures by defining an NFV-node market environment. In particular, an SDN controller acts as the market orchestrator/regulator, assigning *prices* to NFIs which are indicative of their workload. At the same time, NFV nodes target the increase of their 'income'. This means that NFV nodes aim to host popular NFIs that result in higher prices. In more detail, when the demand for a service increases (declines) the price of the service NFIs rises (decreases) accordingly, which in turn may drive NFV nodes to instantiate (consolidate the) NFIs of the corresponding service. In addition to NFIs instantiation/consolidation, each NFV node is also responsible for taking flow steering and redirection decisions.

In DRENCH the market orchestrator is setting the control parameters of *i*) *minimum NFI price* and *ii*) *off path penalty factor*. The minimum NFI price defines a threshold for consolidating NFIs whose prices are below the minimum one. Since NFIs' prices are representative of their workload, the minimum NFI price indicates the threshold below which an NFI is considered being *under-utilised*, thereby controlling the number of active NFIs. On the other hand, the off path penalty factor controls the path-stretch of a flow in the context of SFC, thereby penalising the choice of NFIs that force the flow to deviate from its shortest path towards the destination. Considering Flow Completion Time (FCT) as an index of flow performance, DRENCH minimum NFI price (off path penalty factor) defines the tradeoff between under-utilised instances (flow path stretch) and FCT.

The main technical contributions of this work involve:

- *A feasible NFV management approach:* in DRENCH resource management decisions are taken locally by NFV nodes while the market orchestrator solves lightweight problems, addressing a complex problem in a computationally feasible way with respect to *i) path-stretch, ii) number of active NFIs per service, iii) load on each NFI and iv) flow completion time.*

- *A decoupled NFV resource management framework:* in DRENCH, NFV nodes do not have to be owned by the same entity, which thereby contributes to the

incremental adaptation of NFV in arbitrary network topologies.

- *Implementation and large scale evaluations:* We prototyped DRENCH in Cloudlab testbed and Mininet to demonstrate that DRENCH is immediately deployable in an SDN environment. With Mininet, we compare DRENCH to a centralized approach Slick [110] on a 4K-Fat tree topology. Additionally, we compared DRENCH in a simulation environment consisting of a Rocketfuel topology (87 switches) to a custom centralized approach: SIMPLE [23] on top of a E2 SDN framework [22]. Our results show that DRENCH is robust to: *i)* asymmetries caused by dynamics of arrival/departure of *elastic* flows with different service needs, and, *ii)* the instantiation/removal/failure of service instances (see Section 9.4).

9.1 Design Overview

9.1.1 Desired Properties

DRENCH is an in-network, congestion-aware, load balancing algorithmic framework that handles SFCs and dynamic NFIs in arbitrary network topologies. In designing DRENCH, we focus on providing the following key properties:

Pt1 **Efficiency:** As an NFI placement mechanism, DRENCH should neither under-utilise nor over-utilise the resources available in NFV nodes.

Pt2 **Cost awareness:** DRENCH should instantiate the minimum NFIs to meet the requirements of the SFCs for the flows through the network at any point in time, and balance the utilisation across the active NFIs.

Pt3 **Fine-grained flow handling:** DRENCH must meet each flows' SFC functional requirements, in terms of end-to-end latency and minimum throughput.

Pt4 **Responsiveness:** DRENCH should react to SFC traffic demand fluctuations, especially when traffic is volatile and bursty [130], [131], for arbitrary network topologies.

Pt5 **Incremental deployability:** DRENCH should require the minimum possible modifications in terms of protocols and network infrastructure. It should also be applicable to any of the existing SDN architectures [22, 111, 121] with minimal changes. Furthermore, it should be possible to directly apply DRENCH to a subset of available switches and of incoming traffic when necessary.

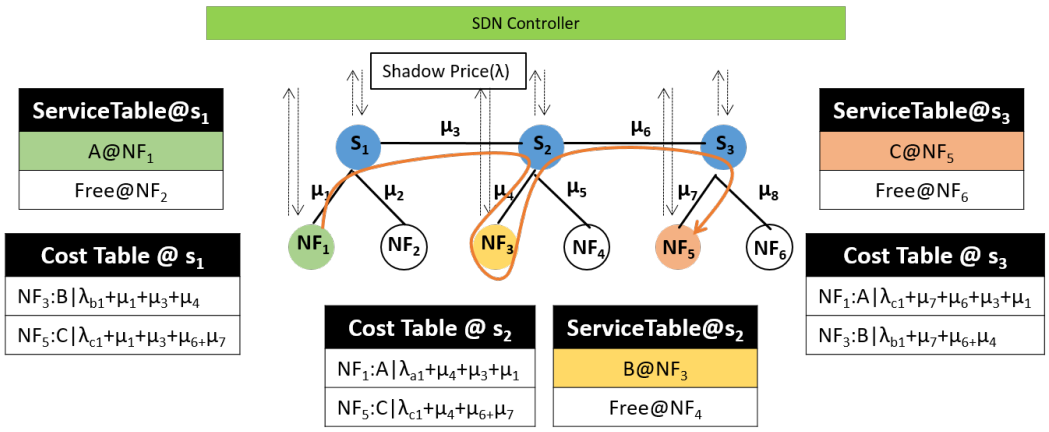


Figure 9.1: DRENCH High-Level Operation

9.1.2 DRENCH Solution Overview

Our framework is designed to leverage the benefits of the centralised as well as the distributed networking paradigms. We use the **centralized approach**, *i.e.,* an SDN controller, to perform tasks with less computation load, but those that need to be carried out in a coordinated fashion across multiple nodes. These tasks include: *i)* gather, compute and disseminate NFI load information periodically to all the decision making entities; and *ii)* set up paths towards instances and egress nodes in case they do not already exist. Additionally, the SDN controller is used to decide which services are applicable to a flow (based on policies and/or flow characteristics). This design choice reduces the complexity of the controller and also overcomes the issues faced by a purely distributed approach, where the decision making entities might not have up to date information, thereby impacting performance. On the other hand, a **distributed approach** is used for decision making at individual NFV nodes. Based on the information provided by the controller, each node independently decides to *i) steer flows* towards the next required service; *ii) redirect flows* to the least loaded instance; and *iii) instantiate/terminate* NFIs in order to adapt to demand. The high-level operation of the proposed mechanism is shown in Fig. 9.1.

9.2 DRENCH Components

DRENCH consists of the following components:

- **Market Orchestrator:** It associates every NFI and link resource to a *shadow price* (*i.e.,* cost) produced by utilising global information available at the con-

troller. The orchestrator regulates the market by allowing the existence of instances above a certain minimum price.

- **Flow Steering and Redirection:** This component steers each flow through a valid sequence of NFIs (according to its SFC) determined by the SDN controller. Steering and redirection takes into account latency, NFI and link costs.
- **NFI Instantiation/Consolidation:** This component instantiates and consolidates NFIs in a distributed way through the market competition between NFV nodes.

Below, we describe each of these components in detail.

9.2.1 Market Orchestrator

DRENCH, as any market-based approach, requires the association of each network resource (commodity), in terms of NFIs and link bandwidth, to an offered price, which is imposed on a given set of incoming flows (demand) that utilise this resource. In particular, when the quantity of demanded resources equals the quantity supplied for a set of prices, we refer to them as *market-clearing prices*. DRENCH market-clearing prices should:
A1) be representative of each NFI's workload,
A2) be derived in the minimum possible time,
A3) not require additional in-network signalling given the existence of an SDN controller [123, 132].
Every price derivation violating requirement (*A2*) and (*A3*) would be in stark contrast with DRENCH desired properties *wrt* responsiveness (***Pt4***) and incremental deployability (***Pt5***), respectively.

DRENCH deploys a *Market Orchestrator/Regulator* component, which by simply exploiting flow path information, already available at the SDN controller, efficiently derives the market-clearing prices; complying to requirements (*A1*)-(*A3*). Inspired by [133], where the authors formulate a Network Utility Maximisation problem (NUM) based on market principles to allocate bandwidth resources to a set of flows, we extend their model to include NFI computational resources. We achieve this by solving the Extended Network Utility Maximisation problem (ENUM) at the Market Orchestrator as we describe next.

We denote the network topology by $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, of $\mathscr{V}$ switches and $\mathscr{E}$ links, where a set of NFV nodes, $\mathscr{H}$, is placed at a subset of switches $\mathscr{H} \subseteq \mathscr{V}$. Then, given a set of NFIs executing a set of $\mathscr{S}$ services and a set of $\mathscr{F}$ flows, we associate each link, $\forall e \in \mathscr{E}$, with a bandwidth capacity, b_e, and each NFI s at NFV node h with b_s^h com-

$\mathscr{G}$	Network topology
$\mathscr{V}$	Set of switches
$\mathscr{E}$	Set of links
$\mathscr{H}$	Set of NFV Nodes
$\mathscr{S}$	Set of services
$\mathscr{H}_s$	Set of NFV nodes executing service s
$\mathscr{F}$	Set of flows
x_f, x_f^*	Rate and optimal rate of flow f
$U_f(x_f)$	Utility function of flow f
b_e	Bandwidth capacity of link e
$a_{e,f}$	Coefficient = 1, if flow f traverses link e
b_s^h	Computational resources of NFI s at h
d_s	Computational power Required by a NFI of s for processing a single bit of traffic
$d_{s,f}^h$	Coefficient = d_s if f is processed by NFI s at h
w_f	Weight of flow f
λ_s^h	Service cost of NFI s at NFV node h
$\underline{\lambda}, \bar{\lambda}$	Minimum and Maximum shadow prices, defining the efficiency of an instance
p_{v_i,v_j}	Shortest path from switch v_i to switch v_j
μ_{v_i,v_j}	Communication cost from switch v_i to switch v_j
$\mathscr{C}_{v_i,h}(s)$	Communication and service cost from switch v_i to a NFI executing service s at NFV node h
$\lvert p_{v_i,v_j} \rvert$	Number of Hops from switch v_i to switch v_j
$\Delta p_{v_i,h}^f$	Shortest path deviation overhead
ρ	Off Path penalty factor
$\mathscr{C}_{v_i,h}^f(s)$	Estimated $\mathscr{C}_{v_i,h}(s)$ cost of flow f including ρ
θ_{rid}	Redirection threshold
P_h	Profit of NFV h in terms of shadow prices
$\widetilde{\lambda}_{on}, \widetilde{\lambda}_{off}, \widetilde{\lambda}$	On-/Off- path and expected competitive price

Table 9.1: DRENCH Notation Description

putational resources, in order to form the ENUM problem that maximises the total utility of the system. Similar to NUM, we associate each flow rate, $x_f \geq 0$, with a utility that is a weighted logarithmic function, $U_f(x_f) = w_f \log(x_f)$, of weight w_f, capturing a decreasing marginal gain as the flow rate increases (*i.e.,* rate changes at low rate flows have a greater impact on their utility). In turn, we maximise the total system utility, $\sum_{f \in \mathscr{F}} U_f(x_f)$, subject to link capacity constraints, $\sum_{f \in \mathscr{F}} a_{e,f} x_f \leq b_e$, $\forall e \in \mathscr{E}$, and computational resource constraints, $\sum_{f \in \mathscr{F}} d^h_{s,f} x_f \leq b^h_s$, $\forall s \in \mathscr{S}, \forall h \in \mathscr{H}$; where $a_{e,f}$ is a coefficient equal to 1 if flow f traverses link e and 0 otherwise, while $d^h_{s,f}$ equals the computational power required by service s for processing a single bit of traffic, d_s, if f is executed at NFI s of NFV node h, and 0 otherwise. Parameters $a_{e,f}$ and $d^h_{s,f}$ describe the path of each flow and therefore they are known to the SDN controller which provides them to the Market Orchestrator.

Since the objective function is differentiable and strictly concave, while the feasible region of the constraints is compact, the optimal rates x^*_f $\forall f \in \mathscr{F}$ exist, are unique, and can be found efficiently by Lagrangian methods. Based on [133], it can be shown that the dual problem of the ENUM is:

$$\begin{aligned} \text{maximise} \sum_{f \in \mathscr{F}} w_f \log \Big(\sum_{e \in \mathscr{E}} \mu_e a_{e,f} + \sum_{h \in \mathscr{H}} \sum_{s \in \mathscr{S}} \lambda^h_s d^h_{s,f} \Big) \\ - \sum_{e \in \mathscr{E}} \mu_e b_e - \sum_{h \in \mathscr{H}} \sum_{s \in \mathscr{S}} \lambda^h_s b^h_s \\ \text{subject to} \\ \mu_e \geq 0, \; \forall e \in \mathscr{E}, \\ \lambda^h_s \geq 0, \; \forall s \in \mathscr{S}, \forall h \in \mathscr{H}, \end{aligned} \tag{9.2.1}$$

where μ_e and λ^h_s are the Lagrange multipliers of link e and service instance s at NFV node h respectively. The Lagrange multipliers are also known as *shadow prices*, due to their association to the optimal rates of each flow:

$$x^*_f = \frac{w_f}{\sum_{e \in \mathscr{E}} \mu_e a_{e,f} + \sum_{h \in \mathscr{H}} \sum_{s \in \mathscr{S}} \lambda^h_s d^h_{s,f}} \tag{9.2.2}$$

where weight w_f is perceived as the budget that flow f is willing to pay for its rate, while the denominator is the cost imposed to the flow in order to use the resources along its path. In that sense, *each Lagrange multiplier can be considered as the price of a particular resource*, leading us to the following definition about communication and service cost.

Definition 9.1 (Communication Cost) The *communication cost* between two switches, $v_i, v_j \in \mathscr{V}$, is the sum of on-path link shadow prices $\mu_{v_i,v_j} = \sum_{e \in p_{v_i,v_j}} \mu_e$, where p_{v_i,v_j} is the shortest path between switches v_i and v_j; while the *service cost* of an instance s at NFV node h is the shadow price λ_s^h.

Note that the service and communication costs are kept in the forwarding tables of the NFIs, *i.e.,* the decision making nodes, and are updated periodically by the SDN controller after being estimated by the Market Orchestrator (see Fig. 9.1).

Shadow prices are indicative of the workload at a particular resource, complying with (*A1*). In fact, from (9.2.2), we derive that the value of a shadow price, λ, defines the maximum possible rate that flows using that resource can achieve, w_f/λ.[26] Based on the maximum achievable flow rate we can define the efficiency of a NFI as a range of shadow prices.

Definition 9.2 (NF Utilization) The Market Orchestrator determines the load of a NFI by a shadow price range $[\underline{\lambda}, \bar{\lambda}]$, where if a service cost, λ_s^h, is less/more than $\underline{\lambda}/\bar{\lambda}$ the NFI is considered under-/over-utilised, respectively.

Given the shadow price range $[\underline{\lambda}, \bar{\lambda}]$ the Market Orchestrator tries to maintain the minimum required number of instances per service type (***Pt2***) by: *i)* terminating instances that are underutilised and *ii)* allowing for more instances for services whose existing instances are over-utilised (see Section 9.2.3).

9.2.2 Flow Steering and Redirection

9.2.2.1 Flow Steering

Given a placement of NFIs and their respective shadow-prices, as determined by the Market Orchestrator, DRENCH's flow steering component is responsible for steering each new incoming flow towards the chain of required services. The flow steering component tries to route the flow through the chain that imposes the lowest possible cost to the flow. Determining the optimal end-to-end path of a flow through the SFC is a NP-complete problem [134]. DRENCH works on a hop-by-hop heuristic basis, picking each time the best next-hop NFI choice, in an effort to achieve instantaneous and adaptive steering decisions (***Pt4***).

[26] It follows that the shadow prices are positive when a resource is totally utilised and 0 otherwise. To introduce a minimum workload to the resources that are not saturated, we add a set of *dummy flows* into $\mathscr{F}$ when solving (9.2.1).

We illustrate DRENCH's flow steering component through the following example. Assume that flow f arrives at the network requiring the execution of service s (or service chain $s_1/s_2/.../s_m$) before being delivered to destination v_f. Let v_i be the switch that has to make a steering decision about f and $\mathscr{H}_s \subseteq \mathscr{H}$ be the set of NFIs of service s. Then the combined communication and service s execution cost at $h \in \mathscr{H}_s$ is $\mathscr{C}_{v_i,h}(s) = \mu_{v_i,h} + \lambda_s^h$. The flow steering component initially estimates the shortest path deviation overhead applied by steering flow f to instance h in terms of hops, *i.e.,* $\Delta p^f_{v_i,h} = |p_{v_i,h}| + |p_{h,v_f}| - |p_{v_i,v_f}|$, weighted by an *off-path penalty factor* ρ. Therefore, the estimated cost applied to the flow for executing service s at NFI of h is $\mathscr{C}^f_{v_i,h}(s) = \mathscr{C}_{v_i,h}(s) + \rho \Delta p^f_{v_i,h}$. Then, v_i selects the next service instance s of f that minimises $\mathscr{C}^f_{v_i,h}(s)$:

$$h^* = \underset{h \in \mathscr{H}_s}{\arg\min} \mathscr{C}^f_{v_i,h}(s) \tag{9.2.3}$$

In Eq. (9.2.3) the off-path penalty factor, ρ, dis-incentivises node v_i from sending flow f away from its shortest path towards v_f. Eq. (9.2.3) applies on a hop-by-hop basis, that is, it is calculated at each NFV node responsible for forwarding flow f towards the next instance in its chain.

Lastly, upon making a steering decision, switch v_i informs the SDN controller that flow f is forwarded towards h^* to execute service s. At the same time, the SDN controller is setting up paths towards NFIs and/or egress nodes as necessary.

9.2.2.2 Flow Redirection of Stateless and Stateful flows

The cost of a service instance might change dramatically throughout the duration of a flow, rendering previous flow steering decisions outdated. Therefore, *redirection of existing flows* is necessary in order to keep the expenditure of existing flows at low levels (***Pt3-Pt4***) and avoid routing through overutilised instances (***Pt1***). We realise flow redirection as follows: if the cost difference between two instances of s at h and h', as seen by switch v_i, is bigger than a redirection threshold, θ_{rid}, $\mathscr{C}_{v_i,h}(s) - \mathscr{C}_{v_i,h'}(s) > \theta_{rid}$, switch v_i repeats the flow steering process for a portion of flows that v_i currently forwards to h. The redirection threshold is set to $\theta_{rid} = \bar{\lambda} - \underline{\lambda}$.

Rerouting of stateful flows to dynamically instantiated services for improving load balancing is usually complex and costly. For instance, solutions such as Split/Merge [72], pause ongoing flows in order to transfer internal NF and forwarding states. In DRENCH, we leverage the approach in Split/Merge [72], to pause ongoing flows and transfer the internal state of the involved network functions. To identify service instances, we make use of the Information Centric Networking (ICN) construct which is proven to be beneficial in terms of providing flexible routing, and reducing the

routing states at the switches [129].

9.2.3 Instantiation

In DRENCH, NFV nodes autonomously provide services in an effort to maximise their profit, in terms of *shadow prices* (*i.e.,* the cost to execute some service). In particular, let S_h be the set of NFIs at some NFV node h, then the profit of h in terms of *shadow prices*, λ_s^h, can be estimated, as:

$$P_h = \sum_{s \in \mathscr{S}_h} \lambda_s^h \tag{9.2.4}$$

DRENCH NFI instantiation/consolidation scheme defines how service demand and NFI *shadow prices* affect the individual NFV node decisions to manage the number of service instances. Through competitiveness, NFV nodes achieve responsiveness to NF demand changes, while the market orchestrator ensures market efficiency, as we explain next.

9.2.3.1 NFV Node Competitiveness

Let the *shadow price* of an NFI s' at NFV node h' be λ'. We are interested in estimating the competitive price of a potential NFI s at an NFV node h, $h \neq h'$, with respect to λ'.

Definition 9.3 (Shadow Price) The *shadow price* of NFI s is *competitive* in the price of NFI s' when the flow steering component has a preference, or is indifferent, of steering new flows at s.

Then, let μ be the communication cost, between NFV node h' and h that are Δp hops away, and f be a new flow that is about to get steered at NFI s'. Then according to DRENCH's flow steering component, the minimum competitive price at s for flow f, would be equal to $\widetilde{\lambda}_{off} = [\lambda' - \mu - \rho\Delta p]^+$, where ρ is the off-path penalty factor.[27] The off-path penalty factor is taken into account as in the worst case that flow f will have to traverse Δp additional hops to reach s from s'. This acts as a disincentive for a node to forward traffic to nodes that are far off from the flow's shortest path. On the other hand, in the best case, that flow f is forwarded to NFV node h' by NFV node h, meaning that s is already *on the path* of flow f and

[27] $[\cdot]^+$ denotes the projection onto nonnegative orthant.

additional hops are not required;[28] the minimum competitive price at s for flow f is $\widetilde{\lambda}_{on} = \lambda' + \mu$.

The expected competitive price of NFI s with respect to NFI s' price λ' will be a value between $(\widetilde{\lambda}_{off}, \widetilde{\lambda}_{on})$. Let y be the total amount of traffic with competitive price $\widetilde{\lambda}_{on}$. Then y can be considered as the local information of NFI s' demand at NFV h that accounts for the utilization percentage $d_{s'}y/b_{s'}^{h'}$. Here, $d_{s'}$ is the computational power required by the service of NFI s' for processing a single bit of traffic and $b_{s'}^{h'}$ is the fixed computational resources that are allocated to NFI s'. Then the expected competitive price is estimated as:

$$\widetilde{\lambda} = (d_{s'}y/b_{s'}^{h'})\widetilde{\lambda}_{on} + (1 - d_{s'}y/b_{s'}^{h'})\widetilde{\lambda}_{off} \tag{9.2.5}$$

9.2.3.2 NFI Instantiation

As long as a NFI shadow price, λ, executing a service at NFV node h, is lower than the maximum target price, $\lambda < \bar{\lambda}$, this service is not considered over-utilised and an instantiation of an additional NFI of the same service at h is prohibited (see also *Definition 9.2*). On the other hand, if $\lambda > \bar{\lambda}$, the Market Orchestrator limits the number of NFI that can be created by competing the NFI with service cost λ to $\lfloor \lambda/\bar{\lambda} \rfloor$. Therefore, given the set of allowed services for instantiation at each NFV node h, h estimates the expected competitive prices of every NFI. Then moving from the highest to the lowest competitive price, the NFV node instantiates the service associated with the price $\widetilde{\lambda}$ as long as *i*) it is expected that the instance will not be under-utilised, $\widetilde{\lambda} > \underline{\lambda}$, and *ii*) the Market Orchestrator maximum number of instances allows it, respecting properties (***Pt1***), (***Pt2***), and (***Pt4***).

9.2.3.3 NFI Consolidation

If the price of an instance is below the minimum target shadow price, $\underline{\lambda}$, the NFV node consolidates this instance ***(Pt2)***. When there is a service availability requirement, the market orchestrator can hinder the consolidation of the last instance of that service.

[28] In practice, it is not the NFV node that is aware of the forwarded traffic but the switch that the NFV node is attached to.

9.3 Implementation

DRENCH's prototype involves the implementation of an SDN controller/orchestrator, Open vSwitches, and a set of custom NFIs. The overall implementation, including the controller extensions for supporting NFIs, is $\sim$ *1800 LOC*.

9.3.1 Control Plane: DRENCH Controller

We extended Pox[29] to serve as DRENCH's controller, where policy specifications are provided as input.

Flow Classifier and Policy Enforcement: DRENCH controller performs fine-grained flow classification, based on the standard IP 5-tuple. As a flow is mapped into a policy specification, the Controller applies the sequence of the involved service functions. We dedicate the combination of *IP-ToS* field and *L4: destination port* to represent the service chain ID and the sequence of functions in the service chain respectively.

Flow Steering: In order to be more flexible, and readily deployable with NFIs, we rely on the switch-based service chain/network function ID mapping that can be supported by openflow switches, without the need for any modifications to the NFI. The controller sets the path from the ingress/NFV node to the next NFV node/destination in the chain. For finer granularity at each ingress or intermediate NFV node, the flow rules are installed based on the match obtained from the flow-classification - this readily enables the capability to match and correlate the packets that enter/exit the NFIs, even when NFIs modify the packet headers, while the rest of the intermediate switches rules are installed for forwarding the traffic via "tunnels" to the next NFV node or egress switch. For tunnelling, we enhance the named instance source routing scheme [129], wherein the tag comprising of the VNFs service type (*SVC-ID*), the switch (*SW-ID*), and the pinned compute core (*CoreID*) uniquely represent the NFI. It is defined as: $NFI\text{-}ID = CoreID|SW\text{-}ID|SVC\text{-}ID$.

This enables the paths to be installed proactively on all the intermediate switches as soon as an NFI is discovered in the network. Our proposal of ID-based tunnelling can be realised by using either the Multi-Protocol Label Switching (MPLS) or VLAN tags, underlay (unused IP/TCP header fields like DS/option fields) or with Network Service Header (NSH) [33]. The choice of Layer-2.5 (MPLS) or Layer-2 (VLAN) tag makes it convenient to define the match in Openflow switches, *i.e.,* by being agnostic to L3/L4 fields allow any TCP/UDP flow to be matched with a same rule.

[29] https://openflow.stanford.edu/display/ONL/POX+Wiki

This helps to significantly reduce the number of switch rules. As most MBs readily support VLAN as opposed to MPLS [126], we make use of Layer-2 tags to realise the NFI-based tunnels.

9.3.2 Data Plane: Openflow Switches and Network Functions

One of the key challenges we faced in the implementation of DRENCH prototype was the unavailability of a MB/NFV-capable switch in the Mininet environment. Our solution was to realise NFIs as hosts connected to the switches. However, this resulted in additional challenges since OpenFlow is designed with a Southbound API control channel between the controller and the network switches, but not the hosts. Therefore, any information (*e.g.,* estimated cost of other instances/shadow-prices) that had to be exchanged between the controller and the NFV-hosts had to be performed either via the switch (*e.g.,* we used LLDP packets to make the controller aware of the presence of NFV-hosts) or via a separate channel.

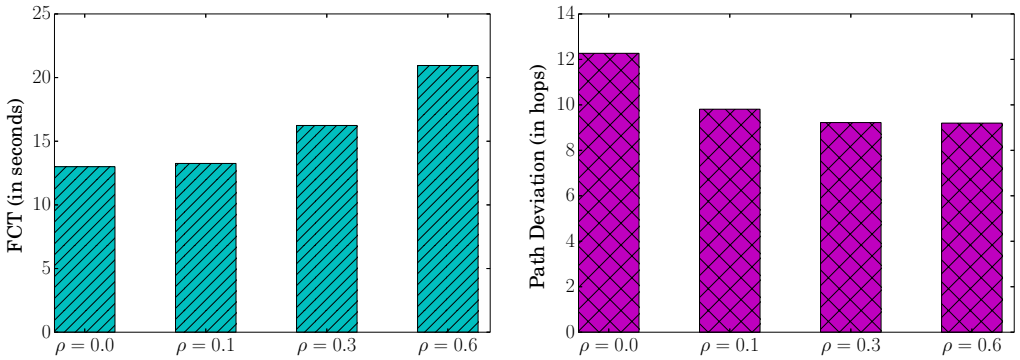
In a real-world deployment, we believe that this would not be the case since NFV nodes will have a Southbound-API based channel from the controller through an in-built switch. Once the communication channels were established, the controller was able to both obtain and disseminate `cost` related information periodically. Moreover, to demonstrate that the NFI capability can be realised on Openflow switches, we implemented on each host a host-local Open vSwitch and controller. The host interface is setup as a port of the local vSwitch. This way, we leveraged the local Open vSwitch and Pox controller to implement the VNF specific functionality, *i.e.,* to monitor and disseminate NFI-specific load information and communicate it to the global controller.

9.4 Evaluation

Goal of our evaluation is three-fold:
i) study the effect of different DRENCH parameters, the resulting trade-offs, and the impact on performance in order to fine-tune DRENCH;
ii) highlight the benefits of DRENCH on a controlled topology (we perform these evaluations on a CloudLab[30] test-bed); and
iii) compare the performance of DRENCH with other approaches in large scale scenarios (both data-center and Rocketfuel topologies).

We make use of the DRENCH prototype on a data-center topology in a Mininet

[30]https://cloudlab.us



(a) Flow Completion (b) Path Deviation

Figure 9.2: Off-path penalty (x-axis)

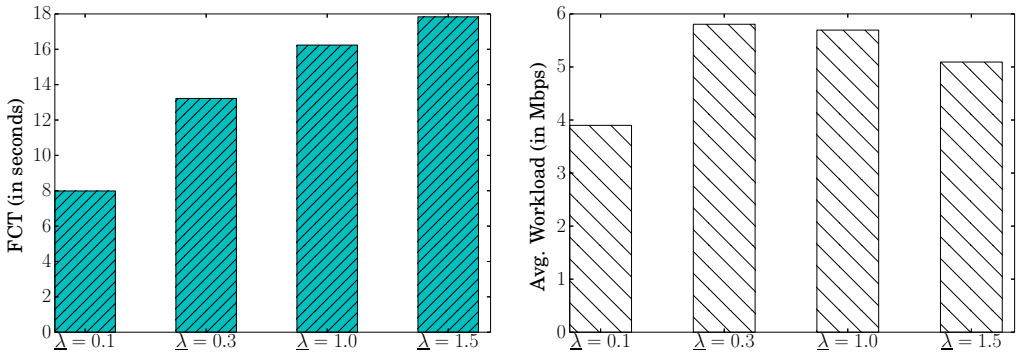
Cluster[31] to study the benefits of DRENCH in terms of Flow Completion Time (FCT), delay, number of active NFIs, NFI utilisation, and the impact of redirection. We compare DRENCH against a centralised approach (*Slick* [110]) and *DRENCH without redirection (DwoR)*.

Moreover, we build a custom flow-level simulator to study the benefits of DRENCH in terms of path deviation, average throughput and FCT in comparison to a custom centralised approach, *i.e., SIMPLE* [23] *on top of E2 SDN framework* [22]. Unlike similar works that focus only on the latency requirements of service chains, we also emphasize on FCT - arguably the most important metric for the user [135].

9.4.1 DRENCH Parameter design and study of tradeoffs

We implemented DRENCH on a python-based discrete event simulator using *SimPy*[32], in order to be able to flexibly fine-tune DRENCH's parameters and to perform large scale evaluation. For these experiments, we perform simulations on a Rocketfuel topology with 27 hosts, that send/receive flows, and 57 nodes that are capable of hosting NFIs. Unless stated otherwise, the off-path penalty, ρ, is 0.3 while we consider SFCs of length equal to 2. We then apply the results of the simulation study to setup DRENCH's prototype.

[31] http://mininet.org
[32] http://simpy.readthedocs.io/en/latest/



(a) Flow Completion (b) Work Load (Throughput)

Figure 9.3: Shadow Price threshold (x-axis)

9.4.1.1 Off-path penalty, ρ

Traditionally, flows follow the shortest path towards their destination, deviating only for traffic engineering reasons and/or in response to link/node failures (*e.g.,* fast-reroute in MPLS). When it comes to service chaining, flows deviate from their shortest path in order to be served by NFIs. In DRENCH the off-path penalty factor, ρ, controls the tradeoff between the shortest path deviation and FCT by *trading* path deviation overhead for less congested NFIs, as Fig. 9.2 indicates. In more detail, the FCT increases function to off-path penalty, ρ, since flows prefer to get served by a more congested NFI (*e.g.,* with a higher service cost) than deviating from their shortest path, Fig. 9.2a. Hence, for our experimental purposes we choose a value of ρ in the range of 0.3-0.5. The exact setting of the ρ factor is up to the network operator. During low-demand periods (*e.g.,* during nighttime), where links are generally less utilised, operators might choose a lower value to improve FCT (*i.e.,* given low link utilisation, extra path deviation should not cause problems). On the other hand, in high demand periods, path deviation should be kept to lower levels, even if this increases the individual flows' FCT, in order to avoid extensive path stretch.

9.4.1.2 Shadow price Threshold

Fig. 9.3 shows the results of varying the minimum shadow-price threshold, $\underline{\lambda}$, at which new instances are spawned. Fig. 9.3a shows that FCT increases function to threshold values. A low $\underline{\lambda}$ of 0.1 results in lower FCT compared to values in the range of 0.3 to 1.5, but at the same time leads to a poor average utilisation of NFIs, *i.e.,* 4 Mbps as seen in Fig. 9.3b. In particular, utilisation is considered as

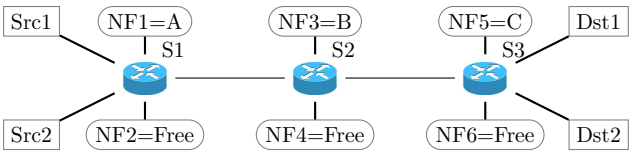


Figure 9.4: Simple Topology with initial placement of NFIs.

Table 9.2: Average Bitrate and Delay

	DRENCH	Baseline
Avg Bitrate (Mbps)	4.033	3.814
Std. Dev of Pkt Delay (ms)	129.151	143.216

the amount of traffic (in terms of throughput - see Fig. 9.3b) that an NFI serves on average. The exact setting of $\underline{\lambda}$ is up to the network operator. If the demand is low, then more instances could be allowed to reduce the average FCTs. In contrast, during high demand periods, the operator might have to compromise on individual flow FCT, in order to make full utilisation of the existing NFIs and eventually serve more flows overall.

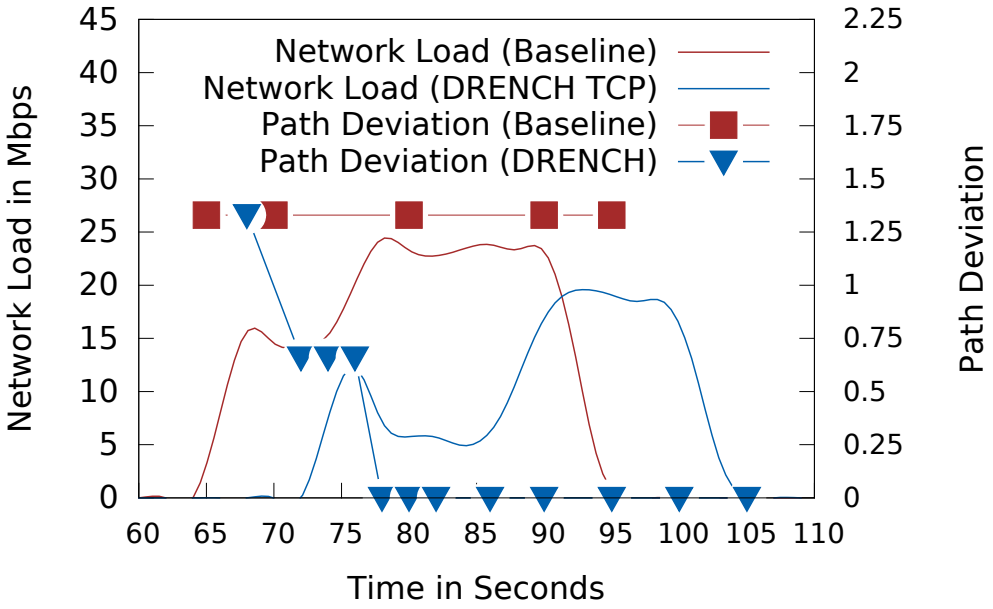
9.4.2 Testbed: Simple controlled experiments

We perform controlled experiments on a small topology, as shown in Fig. 9.4, in our CloudLab setup to illustrate the benefits of DRENCH components. Consider flows *Src1-Dst1*, requiring the service chain of $C/B/A$, which also comprises the ideal placement.

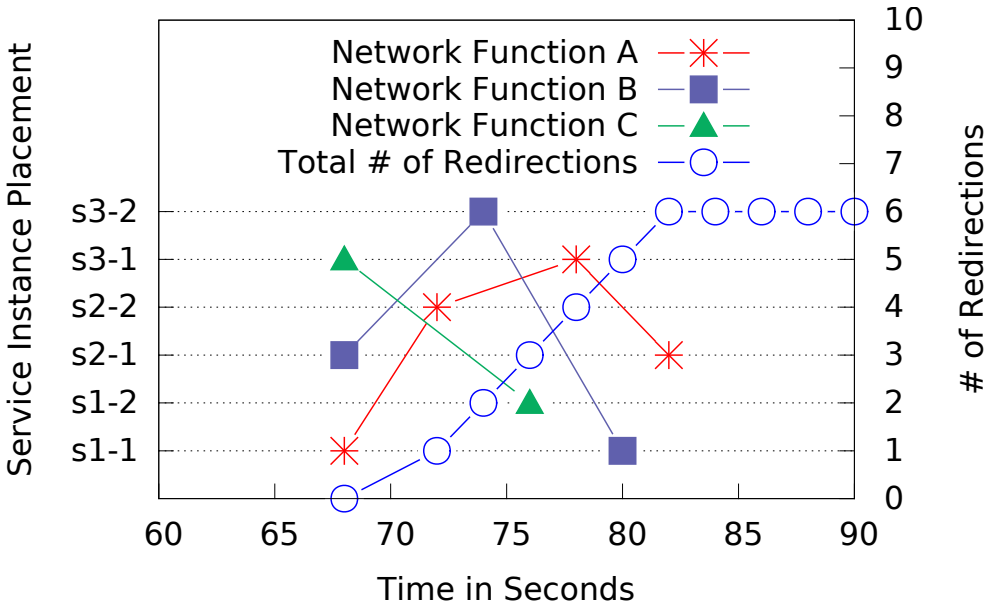
We study the performance and behavior of the system in the worst case scenario, where the services are initially placed in the reverse order as depicted in Fig. 9.4.[33] To prevent an increase in the flow's path length (by going back and forth in this topology), it is desirable to relocate the NFIs (from $A-B-C$ to $C-B-A$ in nodes S1-S2-S3, respectively) to minimize path stretch and FCT.

In Fig. 9.5a, we observe that, initially, due to the service instances being located in the wrong order (*i.e.,* $A-B-C$, instead of $C-B-A$) the flow suffers higher path-stretch, resulting in additional delay and higher network load. Later, as the switches gradually adapt towards the ideal placement, path stretch declines and network throughput increases (see Table 10.2) compared to a non-reactive (Baseline) approach.

[33] We set the NF capacity and the off-path penalty factors to just exceed the threshold required in order to allow for service instantiation.



(a) Network Utilization



(b) NFI Relocation

Figure 9.5: TCP flow with service chain of $C|B|A$

When all the instances individually seek to be competitive and maximise their utilisation, the switches with NFV capacity near the ingress switches end up hosting most of the services. From Fig. 9.5b, we see that DRENCH converges to this ideal case by instantiating and placing the services in the chain along the path towards the destination.

9.4.3 Large scale Evaluation: Data-Center Topology

We setup a Mininet Cluster on Cloudlab to study the performance of DRENCH in a large network topology.

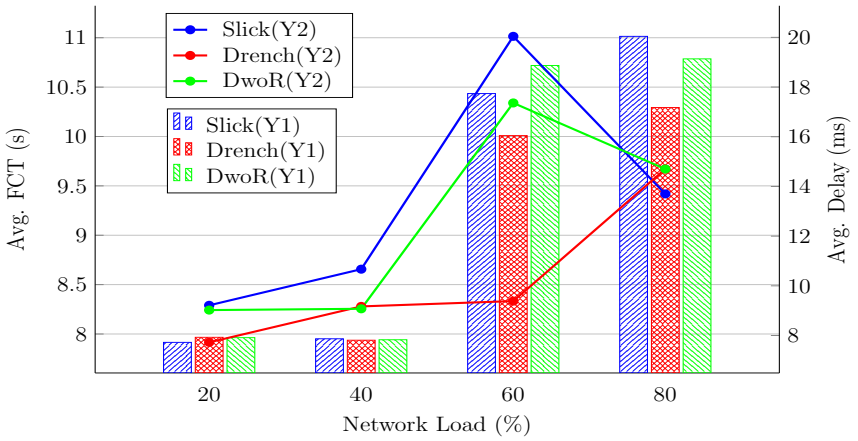
Topology: We use a 4K Fat-Tree topology to evaluate DRENCH in a data-center environment and compare it to Slick (used as an example of a centralized approach.) We consider that only aggregation-layer switches in the fat-tree topology have NFV capability and dedicate 2 cores per aggregate switch for instantiating the NFIs.

Workload: We model the traffic based on the available data center workload characteristics similar to the ones used in [123,130]. The workload constitutes a mix of elephant flows (20% with flow size $\geqslant$ 10MB) and mice flows (80% with flow size $<$ 2MB). Thus, elephant flows account for more than 80% of the traffic bytes. We use iperf[34] and D-ITG [136] to generate traffic with varying network loads. Flows originate from one of the servers connected to a leaf switch and terminate at another server connected to another leaf switch (in either the same or different pod). We use TCP flows in a client/server model with random flow arrival times. Based on the information on the service chaining policies in [137] and the details presented in earlier work (*e.g.,* [23]), we setup a service function chain of 3 distinct Network Functions (NFs). We assume that in total there are 6 NFs available in the network and that each flow requires exactly three of them. The service functions are chosen based on a zipf-distribution with exponent set to 0.3.

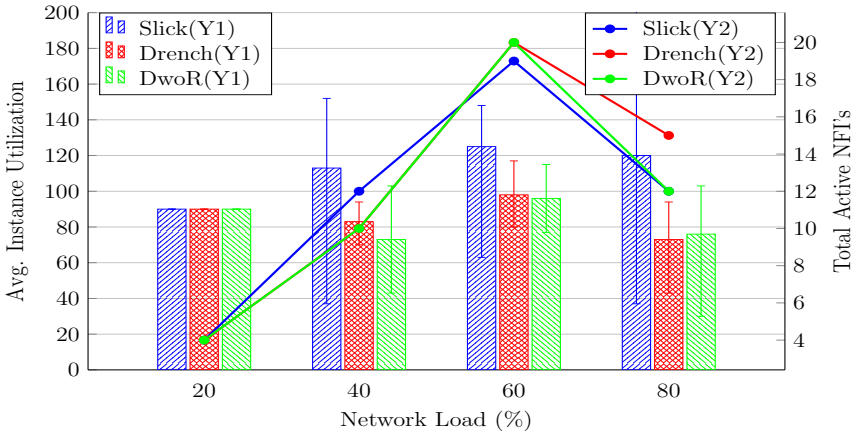
Parameters: Based on our findings from Section 9.4.1.2, we set the `off-path penalty factor` eqaul to 0.5, and the instance `shadow-price` that influences service consolidation and service instantiation decisions as follows. An instance is consolidated when the `shadow-price` reflects a NFI utilisation of $<$ 30% and the flows served by that instance are less than 5. This is done in-order to mitigate the number of flow re-directions and the packet re-ordering impact of flow re-directions. A new NFI is instantiated when the `shadow-price` reflects NFI utilisation of $>$ 85%.

Comparison: In order to perform a fair comparison, we implemented a greedy heuristic-based flow steering approach as in Slick [110] - as a representative of a

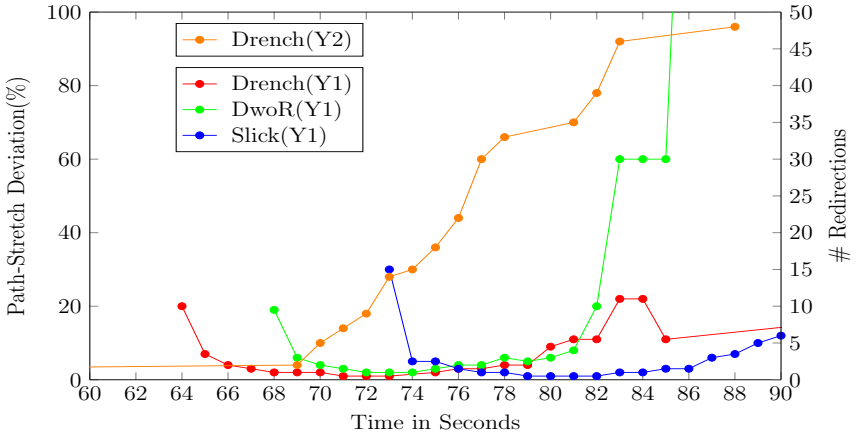
[34] https://iperf.fr/



(a) Overall Avg. FCT and Delays



(b) Avg. # of NFIs and an NFI's Utilization



(c) Impact of redirection

Figure 9.6: Study on a Data-Center Topology (Y1: Left Y axis, Y2: Right Y axis)

fully centralised state-of-the-art load balancing scheme.[35] We also compare against DwoR to study the effects of redirections and its impact on DRENCH. Our main goal is to evaluate the behaviour and performance of DRENCH *wrt* NFI placement, flow steering, and load balancing in terms of its efficiency in NF resource utilisation and FCT.

Figure 9.6a shows the average FCT and packet delays for different schemes. We observe that at lower loads (20-40%) all the schemes have similar FCT, but DRENCH and DwoR incur relatively lower packet delays. However, as network load increases, DRENCH performs better than Slick and DwoR in terms of FCT roughly by 10∼20% . Furthermore, we also observe that DwoR provides better FCT than the centralised approach, while DRENCH outperforms both DwoR and Slick. Finally, we also see that the average delay in the case of DRENCH remains low in most cases and close to the other solutions when the network load is extremely high (80%).

In Fig. 9.6b we see that the average number of NFIs for all the schemes is almost the same. However, DRENCH balances the load more efficiently since the variation in the load among the NFIs and the NFI utilisation is maintained at low levels.

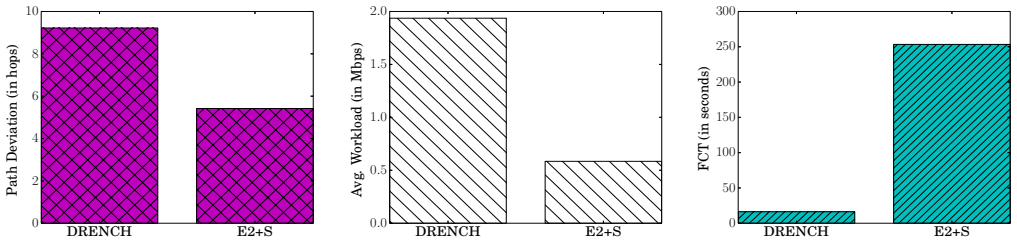
In Figure 9.6c, we present the specific case for network load of 80%, where we observe that DRENCH is able to get close to Slick, which is optimal in terms of path-stretch deviation. We can also see the benefits of DRENCH redirections in order to correct path-deviation during traffic bursts (see 70-80secs) and also when a large number of flows terminate. In both cases we can see that DRENCH is more effective than DwoR, in terms of keeping path-stretch at a minimum, providing better load-balancing across NFIs and achieving better FCT.
Note: In DRENCH re-directions enable to reroute flows through lightly loaded links and NFIs, thus aid to lower packet delays. However, the FCT might still get affected due to interim packet-reordering, which result in false congestion signals. We believe that a modified TCP stack as in [138] could help mitigate the packet-reordering issue.

9.4.4 Large scale Evaluation: ISP Topology

To further examine the capability of DRENCH to efficiently use NFIs in a typical WAN ISP environment, we performed simulations with *SimPy*.

Experimental Setup: We performed simulations on the Rocketfuel AS-1755

[35] We implemented the shortest weighted path-based flow routing scheme and not the entire Slick runtime [110]



(a) Path Deviation (b) NFI Work Load (c) FCT

Figure 9.7: Comparison of Drench vs. E2+SIMPLE

(Ebone in Europe) topology.[36] In order to perform a fair comparison, we implemented a greedy heuristic for flow steering which we call `E2+SIMPLE` that uses a combination of E2 [22] for both service instantiation and service-chain path definition, and SIMPLE [23] for flow steering. Note that in terms of performance, the combination of E2 and SIMPLE performs much better than any of the E2 or SIMPLE alone and therefore is the best choice for comparison.

DRENCH vs. E2+SIMPLE: Fig. 9.7 compares DRENCH and E2+SIMPLE (E2+S) approach. We observe that DRENCH presents higher path deviation (see Fig. 9.7a) since it deviates from the shortest path in search of non-congested NFIs. In doing so, DRENCH is able to make better use of the available NFIs (see Fig. 9.7b) and performs instantiation or consolidation when necessary. This way, DRENCH ensures that all the available NFIs are running close to peak utilisation (in terms of throughput served by the NFV nodes) as seen in Fig. 9.7b. With these design choices, DRENCH achieves significantly lower FCT (see Fig. 9.7c). In summary, DRENCH provides a staggering 10× improvement in FCT and is able to support roughly 4× higher workloads, while incurring an average path deviation penalty of up to 2× in comparison to the E2+SIMPLE approach. This extra path stretch though is compensated in terms of higher NFI utilisation and in turn, lower FCT.

Summary of Evaluation: To summarize, our evaluation demonstrates that with its hybrid centralised-decentralised decision-making structure, DRENCH can dynamically load-balance traffic and allocate resources according to demand. DRENCH makes informed decisions on the load of NFIs and accordingly instantiates new or consolidates existing NFIs. In turn, traffic is load-balanced (through flow steering and redirections) to the appropriate instance achieving significantly lower FCT.

[36] `http://www.cs.washington.edu/research/projects/networking/www/rocketfuel/interactive/1755eur.html`

9.5 Conclusion

We have developed a hybrid algorithmic framework for resource management and traffic load-balancing among virtual NFIs that elegantly combines distributed decision-making with centralised control for orchestration and coordination, while performing complex, dynamic service function chaining. DRENCH is designed to dynamically adapt and balance resources utilisation to traffic demand. DRENCH builds on a market-inspired, competition-based `shadow-price` that is used for taking decisions on flow-steering, flow-redirection and service instantiation/consolidation in a distributed manner. A centralised SDN controller performs market orchestration, dissemination of price information and coordination.

Our novel *semi-distributed* approach for dynamic service instantiation and direction of new and existing flows to the least loaded NFV node increases the throughput from each NFV node and in turn reduces FCT significantly. With the help of a prototype implementation on CloudLab and extensive simulations, we illustrate the benefits of using DRENCH, namely that traffic is dynamically load-balanced among instances and the path deviation of flows across the NFIs is kept to a minimum. Resources are efficiently utilised by timely consolidation of NFIs when they are lightly loaded. Overall DRENCH results in almost a 10× reduction in FCT in some of our experiments.

Chapter 10

Routing for Service Function Chains: Neo-NSH

Contents

10.1 Introduction

The key to realizing agile and elastic network functions is the ability to dynamically instantiate, remove and relocate the network functions. Any such activity would result in having either a new set of service paths or the invalidation of existing service paths. The current NSH draft defines a 24-bit Service Path Identifier (SPI) and 8-bit Service Index (SI). SPI defines one of the possible instantiations (a logical path to sequence of service functions that includes one of the several instances of each service function) for a given SFC, while the SI indicates the location within the service path. Typically the order of relation between service chains and service paths is $1:n$ and it grows exponentially [139]. Although 24 bits is large enough to accommodate any sets of possible service paths the complexity is in managing the SPI labels and updating labels to the Classifiers and different SFFs.

10.1.1 Control plane Functionality

In conjunction to the role of control plane listed in the SFC architecture [41], in order for NSH to realize a service plane, the control plane needs to perform at least the following tasks:

- For all the SFCs, construct the map of SPI (labels) needed for each valid logical path to the SF instance in the chain.
- Disseminate (communicate and update) the SPI information to the SFFs and Service Functions (SFs).

Note that the SPI labels would change every time there is either an addition or deletion of as service function instance or changes to the physical topology.

Though, it can be argued that topological changes are rather rare, and addition or removal of service functions too are rather infrequent, it must be noted that key benefit of NFV is in that it enables to realize elastic service function instances that can be dynamically instantiated or de-commissioned to better adapt to the traffic requirements and meet the SLA requirements from customer perspective and also improve on the overall network utilization. Hence the second aspect cannot be ignored for providing a truly elastic and dynamic service function chaining.

10.1.2 Control plane Overhead Analysis

From section §8.2, we observe that the identification of service paths and the classification of different service paths and service chains can be broadly categorized into four different categories.

- ***Category #1:***that assign label for each service type and let the SDN controller to steer traffic in a service by service fashion to different service function instances in the network [129].
- ***Category #2:***that assign label for each service instance and let the SDN controller and SFFs to determine and steer traffic to different service function instances in the network [37].
- ***Category #3:***that assign the label for each service chain - a high level policy of desired service functions and let the network to employ some overlay/underlay to steer traffic across service functions [127].
- ***Category #4:***that assign the identifier to each of the logical service paths to the

service function instances in the network [33].
The scale of required identifiers to define an end-to-end service function chain and the involvement of SDN controller vary for each of these approaches.

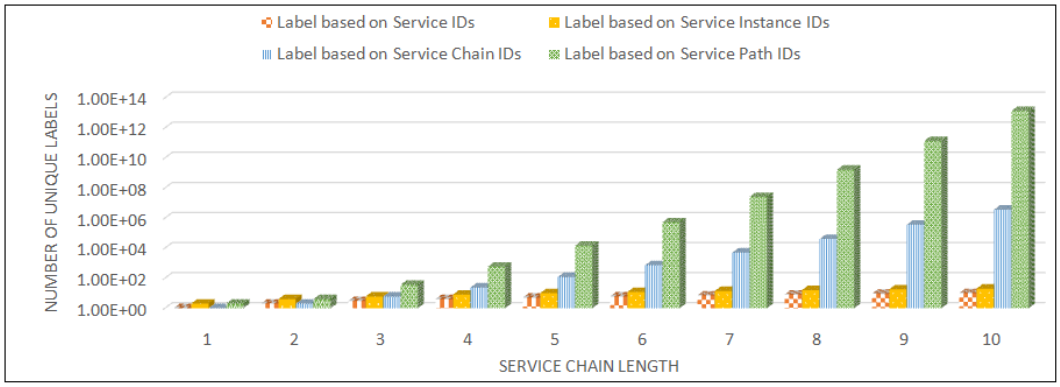


Figure 10.1: Number of Unique Labels for different SFC approaches with varying SFC length

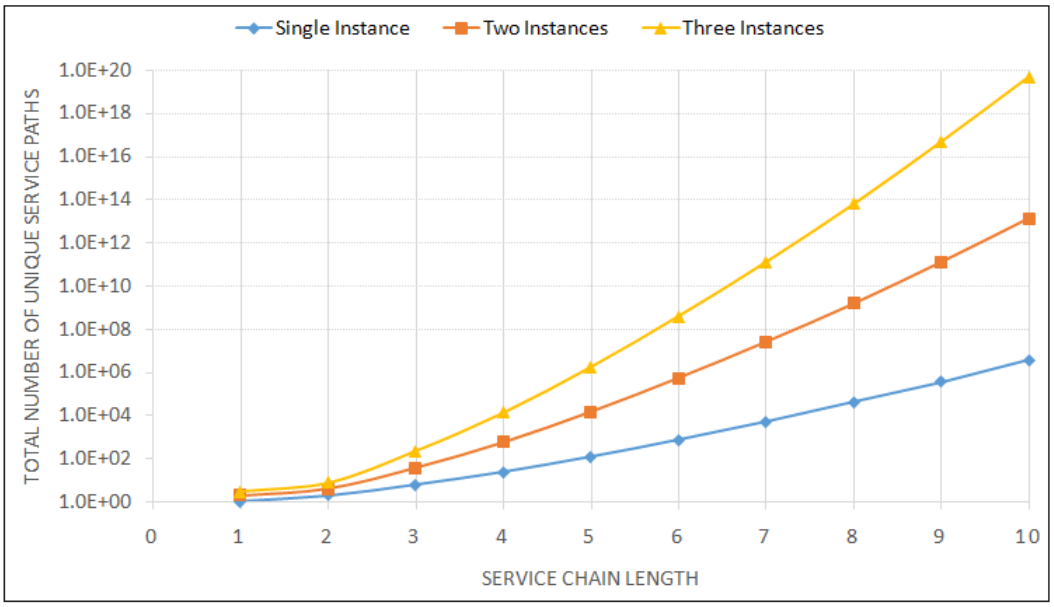


Figure 10.2: Service path IDs for varying SFC length and service instances

We first present the scale of the number of unique identifiers (labels) required for the different approaches discussed earlier. From Figure 10.1, we see that approaches

that utilize the service or service-instance[37] based labels like FCSC [129], NSN [37], and source-routing [139] require a minimum number of labels, while the service-chain ID[38] and service path ID based approaches require a far larger number of labels. Table 10.1 shows the way in which the number of unique labels for each of the different approaches are determined.

Table 10.1: Identifier requirements for different SFC approaches

SFC Approach	Number of Unique Identifiers
Service ID	α *Num. of Function Types* (*SFT*)
Service Instance ID	α *Num. of Function Instances* (*SFI*)
Service Chain ID	α *Factorial*(*Chain Length*)
Service Path ID	α $Factorial(Chain\ Length)^{SFI}$

We observe that the number of active service function instances (SFIs) affect on the number of identifiers required for the approaches relying on the service instance id and service path ids. In the latter case, we can see that number of identifiers scale almost exponentially, as each instance addition results in multiplier of factorial of new possible paths. Analytically, we can show that for addition of every service function instance to each of the service functions types the label requirements for a given chain length grows as shown in Table 10.1.

Given N Instances per ServiceFunctionType

$$TotalNum.\ of\ IDs\ =\ Factorial(Chain\ Length)^{N}$$

Also, note that from Figure 10.2, with the increasing length of service chains and scale of service instances, the number of SPIs scales exponentially.

This implies that burden on the control plane for performing the aforementioned tasks for any addition or deletion of service instances is significant. *i.e.,* with the increase in number of services in a chain and number of instances, the control plane overheads in managing and adapting to the increased number of paths, and dissemination of path information to each of the instances grows exponentially.

In the subsequent sections, we present our proposal Neo-NSH and elaborate on

[37] Service IDs correspond to number of unique services in the network, and service instance IDs to the number of instances for each service type. Figure 10.1 considers two instances per service function type.

[38] We acknowledge that SFC-IDs are mapped based on the required policy intents and are often much limited than actual possible combinations of service chains.

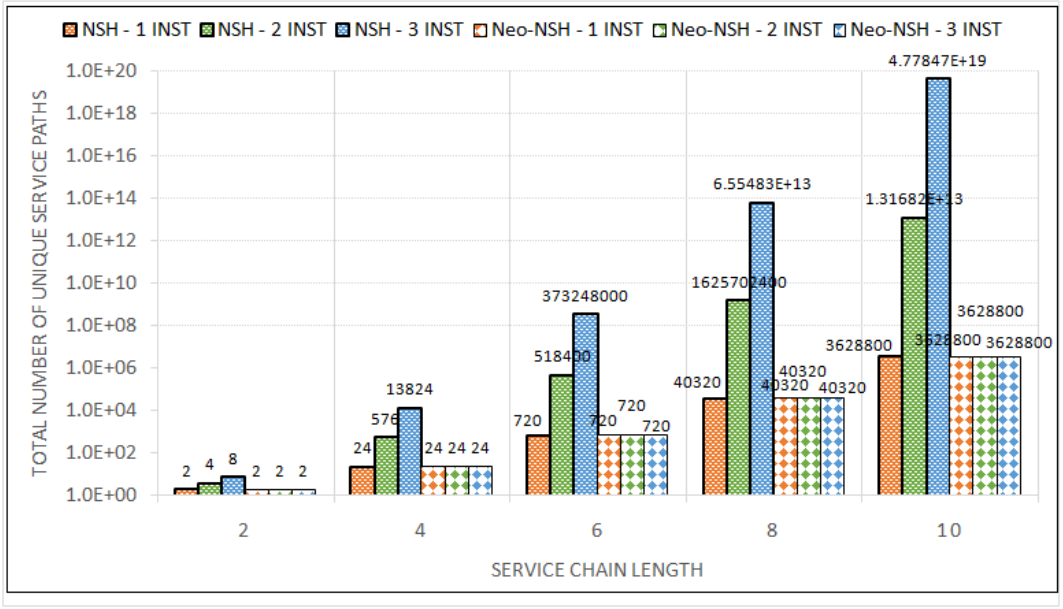


Figure 10.3: Total Service Paths for varying service chain length and instances per service

how it aims to simplify the management and orchestration of Service Path IDs and augments to the base NSH proposal.

10.2 Neo-NSH Proposal

We make the following fundamental observations:

- The service-chain IDs directly reflect the high-level specification of the policy intent, and are relatively easier to program and manged by the network administrators than chaining approach using either the service IDs or the network function instance IDs.
- The number of labels required to support service IDs, and service-chain IDs is minimal compared to the path identifiers.
- service ID base approaches are not affected by the service instance dynamics (i.e., addition, deletion and relocation).

Therefore, augmenting such a feature in NSH, naturally makes it a more robust candidate for implementing a dynamically scalable SFC. Also, we note that the real benefit of SPI (as purposed in NSH) is in providing the traceability and end-to-end

visibility of the service function chains logical path.

We propose to relax the usage of SPI in its true sense, *i.e.,* to refer to the service paths and instead we propose to re-purpose the usage of 24-bit SPI field to denote the service-chain ID. This way SPI more closely reflects the intent of the service chain policy and represents the list of service functions and not the logical path to the service instances in SFC.

In Neo-NSH approach, the Service Forwarders use the SPI and SI fields to represent the service function type rather than the service function instance, and let the network to dynamically choose the best instance based on the service function type (or service name) and the context data information. Service Function Forwarders that select the service path have to rely on either the control plane or the intelligent data-plane to choose the appropriate service instance. And, the role of service classification functions that update the service header is changed to inserting the service-chain ID, while the role of service functions and SFF is unchanged compared to the NSH.

The only down-side of our proposal Neo-NSH is the loss of end-to-end path visibility. The actual path *i.e.,* the list of physical SFs chosen for given SPI cannot be determined statically, as the same SPI could map to different logical paths (path to different service function instance) and physical paths. As the SPI is not static but determined at the run-time (meaning it can change dynamically), it provides an additional benefit of providing the ability to adapt to the network requirements dynamically.

Table 10.2 shows the feature comparison between NSH and Neo-NSH. We can observe that Neo-NSH retains all the key features of NSH. In addition, it augments NSH to be more economic, efficient and scalable solution for SFC,even with the compromise of the loss of end-to-end path visibility. We can observe that overall role of all the involved NSH components remains unchanged.

10.2.1 Dynamic Service Function Instance selection

Typical to the SDN, whenever the flows are classified and service chain ID that determines the policy intent is derived, the controller must determine the appropriate service function instances and then ensure to set the forwarding rules at each of the forwarding elements so that the flow or class of flows traverse through the identified set of service instances. In-order to avoid the path setup latency, forwarding rules can be proactively setup at the forwarders and controller can modify the rules based on the network load in order to re-route and distribute the flows evenly

across different service function instances. The key benefits of this approach over static path identification include: i) the capability to load-balance the traffic across service function instances. ii) resiliency to the addition/removal of service function instances, as it does not affect the SPI. Hence the network becomes more agile and as well it eliminates the need to compute and communicate the SPI labels to the SFFs for any changes affecting the overlay service topology that NSH relies on to forward the packets.

Table 10.2: Salient features of NSH and Neo-NSH

Features	**NSH**	**Neo-NSH**
Topological independence	✓	✓
Transport agnostic	✓	✓
Meta-data sharing and re-classification	✓	✓
End-to-End path visibility	✓	✗
Flexible service instance selection	✗	✓
SPI management overhead	High	Low
Communication overhead	High	Low

In Neo-NSH, by separating the logical service chain from actual service path, we can achieve significant improvements to NSH in terms of:

- **Adaptability and efficiency**: by reducing the control overheads in managing the path IDs and disseminating them to all the service instances.
- **Flexibility**: classifiers, service functions and proxies only need to care about logical service chain and not the service paths.
- **Scalability**: can easily accommodate more instances of service functions without impacting the SFFs, and the reduced forwarding rules ensure to utilize the TCAM space more judiciously and enable to accommodate more flows.

10.3 Preliminary Analysis and Evaluation

We performed preliminary evaluation of the proposal using the mathematical model to demonstrate the benefits of Neo-NSH. We compare Neo-NSH with base NSH approach. In this evaluation, we primarily focus on the demonstrating the benefit achievable in terms of the reduction in the number of service path identifiers in the case of increasing service chain length and increasing number of instances.

10.3.1 Key Benefits

From Figure 10.3, we can see that in case of base NSH, the number of Service Path Identifiers scale exponentially with the increasing service chain lengths. Also for a given service chain length, NSH exhibits exponential growth with the increasing number of service function instances of the chain. With our proposal Neo-NSH, the number of SPIs are not affected by the number of service instances, but increase only with the increasing service chain lengths. Thus Neo-NSH in-comparison to NSH, not only results in significantly lowering the number of path identifier labels, but can support the elastic network functions wherein the network functions can be dynamically instantiated without any additional overhead of updating the SPIs to all the participating NSH aware components of the network. This reduction helps further to substantially reduce the complexity and overhead at the control plane with respect to the computation and communication costs required for managing and disseminating the SPIs to the NSH aware components in the network for any given topology.

10.3.2 Impact on component roles

Table 10.3: Role based comparison for different components in NSH and Neo-NSH

Component	NSH						Neo-NSH					
Servicetype	Insert or Remove NSH		Select Service Function Path	Update NSH		Service Policy Selection	Insert or Remove NSH		Select Service Function Path	Update NSH		Service Policy Selection
Action	Insert	Remove		Dec. Service Index	Update Context Header		Insert	Remove		Dec. Service Index	Update Context Header	
Classifier	+	+			+		+	+			+	
Service Function Forwarder		+	+					+	+			
Service Function				+	+	+				+	+	+
SFC Proxy	+	+		+			+	+		+		

As Neo-NSH keeps the NSH intact, and only amends on the way SPI is used for

routing the packets, We emphasize that the there is no impact on the roles of each of the NSH components and the overall role of all the involved NSH components remain unchanged between Neo-NSH and NSH.

10.4 Conclusion

We have characterized and analyzed the benefits and challenges with the current NSH. We have proposed ***Neo-NSH***, an enhancement to NSH and demonstrated the benefits of our proposal which enables for a more agile and flexible network for service function chaining with elastic network functions. Hence, with SDN enabled networks and NFV based middleboxes, Neo-NSH makes a case for more efficient and scalable realization of dynamic service function chaining.

Chapter 11

Future Prospects

In this chapter we present the possible areas of extension of our work. We consider and present the future prospects of the proposed network resource management and orchestration frameworks.

Contents

11.1 Recap of NF chaining orchestration framework

Table 11.1 compares the supported the characteristics of DRENCH with the state-of-the-art works. We can note that DRENCH is the first work that presents a semi-distributed architecture and tackles both NFI placement and flow steering problems in arbitrary topologies.

11.2 Applicability of DRENCH in other NFV Platforms

DRENCH is an algorithmic NFV orchestration framework for *Service Function Chaining* (SFC). It can be easily incorporated into other NFV platforms. We briefly discuss the key APIs that need to be accounted to port over DRENCH to other NFV platforms.

Southbound interface: We utilize the standard OpenFlow 1.0 APIs for communication with the SDN controller as the southbound interfaces for traffic engineering

Table 11.1: Comparison of related state-of-the-art solutions with DRENCH for desired NFV orchestrator and Management features.

Features	**SIMPLE** [23]	**Slick** [110]	**E2** [22]	**DRENCH+NeoNSH**
Elastic Scaling	Supported	Supported	Supported	Supported
Placement	Supported†	Supported	Supported†	Supported
Steering	Supported	Supported	Supported	Supported
Redirection	Not Supported	Not Supported	Not Supported	Supported
Load balancing	Supported‡	Not Supported	Supported‡	Supported
State Synchronization	NA	NA	NA	Not Supported
Architecture	Centralized	Centralized	Centralized∗	Distributed
Load Balance solver	ILP solver	Heuristics	Heuristics	Heuristics

†: Only offline(static) placement is supported.
‡: Non-adaptive load balancing, depends on the initial estimated traffic.
∗: Initial placement and traffic steering decisions are made centralized controller.

purposes. Also, we extend on the OpenFlow 1.0 to query and update the shadow-price information for each NFI, which can be easily accommodated as Openflow extension APIs. The NFIs do not need any additional APIs.

Northbound interface: The north bound interface abstracted in the form of set of configuration files that are loaded in the controller at the time of module initialization. The configuration files provide the node capabilities in terms of number of cores available on each node to host the NFs. This is a standard procedure across most orchestration frameworks.

As shown in the Figure 2.2, we can see that in ETSI NFV-MANO architecture, this information is stored as distinct database/catalog. In DRENCH, the NFVOs and SDN controller components do not need any additional APIs for communication.

Routing for SFC: Our work NoeNSH strictly adheres to the model of NSH and relies on encapsulation header dictated by the NSH RFC [33] to facilitate the service plane for routing.

Also, the management of routing plane, *i.e.,* enumerating and indexing of service paths are strictly the functions of SDN control plane, and accounts to the logical extension of control plane functionality only. Hence, there is no impact or any changes needed for effecting NeoNSH in any of the data-plane elements.

11.3 Current Limitations and Prospects of Extensions

Considering our current design and implementation of DRENCH and *NeoNSH*, we plan to extend our work in the following aspects:

- Our flow redirection currently accounts to redirect only the flows from one NFI to the other. This works fine for the stateless NFs. But for the stateful NFs, the state maintained at the NFI also needs to be migrated. We plan to incorporate Stateful NF migration alongside flow-redirection to ensure correctness of flow specific state being preserved across the NFI.

- In our Orchestration framework, the NF Placement decision accounts dedicated CPU core, *i.e.,* NFIs are assumed to be pinned to one or more cores and we do not consider or allow the same core to be shared by multiple NFIs, which is true for most of the VM based NFIs. This prospect limits scalability and resource utilization. As, we have shown in earlier chapeter §5, with the container or process based NFIs we can efficiently multiplex multiplex NFI on singe core. Hence, we plan to extend and adapt our optimization model and Orchestration framework to facilitate resource sharing.

- We plane to integrate the flow steering mechanism of *NeoNSH* in DRENCH. This will enable DRENCH to scale beyond the current limitation of supporting 4094 *Network Function Instance*s (NFIs)[39]. We depend on a suitable prototype of NSH in control-plane to accomplish this task.

[39]This limitation of 4094 NFIs comes from the 12 bit VLAN's index used in DRENCH to facilitate name-based service forwarding.

Part III

Addressing NFV Failure Resiliency: High Availability, Fault-Tolerance and Disaster Recovery

Chapter 12

Problem Statement

In this Chapter, we present the need to facilitate NFV failure resiliency in order to provide *Service Continuity* (SC) and *High Availability* (HA) of VNFs in the event of different failures and infrastructure disasters. We outline the related challenges that exist and need to be addressed to achieve Failure resiliency for SFC.

Contents

12.1 Introduction

Network Function Virtualization (NFV) implements network services and middlebox functions such as load balancers, firewalls, NATs, caching proxies, *etc.* in software which can then be run on off-the-shelf commodity servers, avoiding the use of dedicated purpose-built hardware. However, an NFV-based data plane must compensate

for the potential lower reliability of commodity hardware [15]. In addition, the presence of multiple layers of software including hypervisors or container libraries, guest OSes, system and application software, increase the chance of software failures.

12.1.1 Need for NFV Failure Resiliency: High Availability and Fault Tolerance

Fault Tolerance (FT) and High Availability (HA) are also important concerns for network services on NFV platforms. There are a number of studies showing that middleboxes fail [140,141] and software failures [142–144] occur often enough to be of concern. Recent work [140] estimates roughly 40% of network failures are caused by middleboxes, and the measurements on network failures by Gill *et al.* [141] indicate that load balancers have the highest failure probability. Nearly a third (31%) of device failures are attributed to software related issues.

The time to recover from a failure and the overheads for providing resiliency depend on the type of failure. For example, a crash in a software component can be quickly detected and recovered locally by the host operating system on the order of micro seconds, while the recovery from operating system failures may take at least a few milli-seconds (e.g., 10ms for lightweight unikernels like ClickOS [47] and Mirage [145]) to reboot and restore the device. Hardware failures such as link and node failures may take seconds or more.

Multiple NFs may be composed into a service chain run on a single node, either as consolidated functions in a single process [22,146], or in a pipelined fashion [46,47]. Of course, scale-limitations may require the service chain to span multiple nodes. Our failure resiliency framework addresses both cases, and we seek to coherently deal with all different kinds of failures, viz., software failures including the failure of an individual NF instance, and hardware failures such as node and link failures, power outage, etc.

Hence we need a NFV resiliency framework that can be account to address different kinds of failures.To address variety of failures, we posit that "*one approach does not fit all*", as the operational latency and resiliency overheads desired to account different failures drastically varies.

12.1.2 Green Energy on the rise

It has been studied that the DC industry accounts to over 30 Gigawatts of energy per year [147], accounting to roughly 21% of energy accounted by Information and Communications Technology (ICT) [26], and the demand keeps increasing every

year. The carbon footprint of a medium 10 Megawatt data center can range from 3,000,000 to over 130,000,000 kilograms of CO2 [148]. Depending on the electric grid region, Power Usage Effectiveness (PUE) improvements can eliminate millions of pounds of CO2 emissions [149]. These factors have led to tremendous increase in the widespread adoption of renewable resources for powering the data centers.

The recent study [27] indicates a phenomenal increase in the investments ($285.9 billion) for harnessing renewable energy, which is more than double ($130.6 billion) the investments on non-renewable energy resources in 2016. It is also noteworthy that the amount of renewable energy generation capacity has increased by nearly 56 percent over last two years. Greenpeace report [26] indicates that already the companies like Apple, Facebook and Google have started adopting renewable energy to power the data centers.

12.1.3 Need for Disaster Recovery plan: Service continuity in the event of Power outages

It has been shown that the Green energy could be used to adequately power a small DC with reasonable degree of reliability [29]. However, despite the growth in green energy investment and generation, the nature of renewable resource based power poses a challenge towards employing them to large DCs for the following reasons:

i) not sufficient to fully power the large data centers,

ii) highly intermittent and unstable [28]; hence pose a greater challenge in adopting them for the large data centers which require stable and sustained power resources in-order to avoid any service disruptions.

Hence, there is a need to study and address the impact of such power outage disasters on NFs and to facilitate mechanism to ensure SC in the event of such disasters.

12.2 Challenges in achieving NFV Failure Resiliency

12.2.1 VNF Diversity: Challenges and Opportunities

VMs are generally the application processing engines, characterized by the application states, whereas the middleboxes or the VNFs cater towards diverse set of use cases and applications. For example, typical network services like NAT, web proxies, *etc.* Security services *e.g.,* firewalls, intrusion detection and prevention, encryption *etc.* and Value added services like parental control, WAN optimizer and

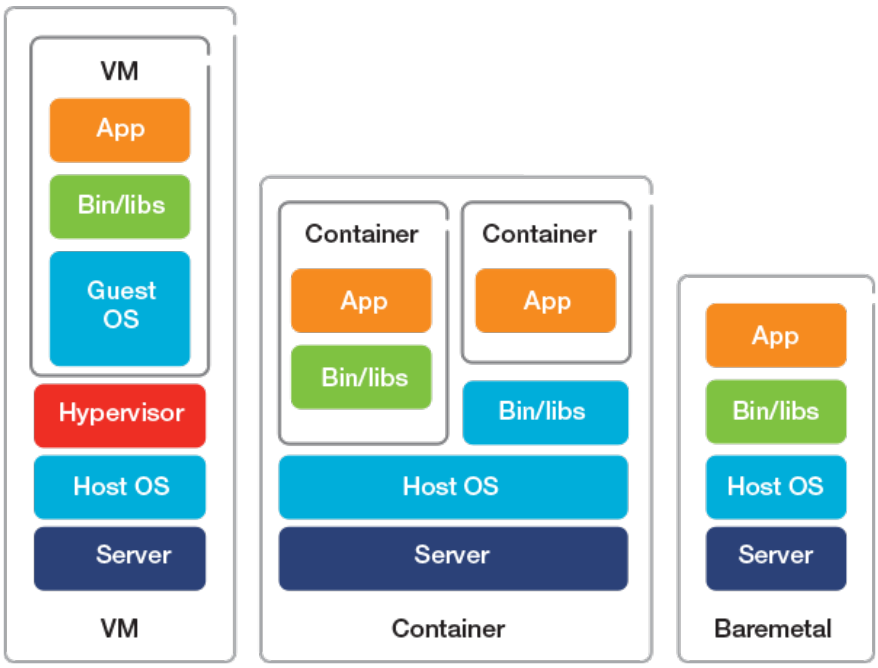


Figure 12.1: Different NFV Deployment Approaches

HTTP enrichment *etc.* . These differences with VNFs result in diverse complexity and processing requirements. Based on the resource (computation, communication, memory and storage) requirements and performance (throughput, latency, jitter, and packet loss) requirements, the ETSI standards [15] broadly classifies the VNF types into four classes namely:

a) data plane functions that exhibit intensive network I/O and demand high packet processing rates,
b) control plane functions that exhibit intermittent network I/O and moderate CPU processing,
c) signal processing functions which are typically CPU intensive, and
d) storage related functions that are heavy on memory and disk I/O.

Nonetheless, VNFs are essentially the high speed packet processing engines that maintain flow/packet specific states and tend to serve millions of packets per second at 10G/40G/100Gbps line rates, and depending on the type of processing, VNFs can be either stateful or stateless. This implies that the frequency at which the VNFs state change can occur is too high and the downtime in the order of milliseconds can lead to severe service disruptions. Hence, to achieve high-availability, consistent updates need to be done more frequently than compared to the traditional VMs.

We also note that, in addition to the VNF state, the network routing state also needs to be updated for reliable processing of subsequent packets. Hence, stateless VNFs only need the routing state update, but the stateful class of VNFs additionally need to snapshot their internal states. Also, the amount of internal state that need to be transferred to back-up the stateful VNF is minimal (order of few bytes to kilo bytes) compared to VMs that range in several giga bytes [72].

12.2.1.1 NFV Deployment Model and Usage Scenario

As shown in Figure 12.1, typical VNF deployment models include

a) ***Dedicated VMs***, *e.g.,* NetVM, OpenNF that allow to instantiate and run the NFVIs as VMs.
b) ***Containers*** or Docker based applications, *e.g.,* OpenNetVM and NetBricks that enable to instantiate and run the NFIs as lightweight containers.
c) ***Packaged network appliances*** *e.g.,* Proxy servers, Load balancers *etc.* that are shipped as isolated portable binaries for specific platform.
d) ***Processes*** *i.e.,* binaries that can be run as dedicated processes on specific OS *e.g.,* . iptables in Linux.

This diversity not only hinders portability - since the Docker based and process based VNFs need to be backed-up on nodes matching the hardware and Operating system requirements, but also pose a challenge towards achieving generalized framework for replication as the needs and means to snapshot and back-up the VNFs significantly differ. However, the promising part of the VNF diversity, especially with the Unikernel based VM's and containers is that the amount of data that need to be backed-up is significantly lowered (order of few mega bytes) compared to the traditional VMs that range in giga bytes.

12.2.2 Service Function Chaining

The flows served by the VNFs are typically subject to more than one network functions, processed in a specific order, *e.g.,* NAT, Firewall, IDS, and Load-balancer. This implies that in-order for the flow/packets to be processed consistently across the replicas, the VNFs cannot be treated in isolation, but the chain (ordered list) of VNFs that a flow/packets go through need to be treated as a group. Hence to maintain the VNF state consistency across the chain, the back-up and snapshot mechanism should consider the periodicity for group of VNFs.

12.2.3 VNF State Anatomy

We leverage the study and analysis of earlier works [72, 121, 150] in discerning the anatomy of internal states. Based on the study of existing VNFs, they can be broadly classified into i) stateless VNFs (those that do not maintain any state for VNF packet processing) *e.g.,* stateless firewalls. ii) stateful NFs - that maintain and store the state for the packets/flows being processed by the NFs. *e.g.,* IDS. Further, the stateful NFs can be sub categorized into i) VNFs with per flow status - that maintain and update states for each of the new flows *e.g.,* Application Delivery controllers and stateful Firewalls ii) VNFs with per packet status - that maintain and update state for every individual packet processed by the VNF *e.g.,* IDS.

In general, the state maintained by the VNFs can be broadly categorized into

i) ***Internal state*** - or the ephemeral state, which do not affect the consistency and may deviate across the replicas,

ii) ***External state*** - typically constitute the flow specific information and counters, which needs to be kept updated across replica, and

iii) ***Coherent state*** - state constituted by the global counters and configuration parameters, which also needs to be kept consistent across multiple replicas [72].

Hence, to maintain the replica, different types of VNFs demand different kinds of state with different levels of state synchronization. We take advantage of this aspect in our work to account for periodicity of state updates.

12.2.4 VNFIs exhibit Non-Determinism

NFs that operate on the same input (flow of packets) can still diverge in their internal states due to i) dependence on hardware whose outcome cannot be predicted, such as hardware clocks, random number generators, *etc.* , and ii) race conditions in accessing shared variables among NF threads. For example, a load balancer (even with the “Active:Active" redundancy configuration) that assigns one server from a pool of backend servers for each TCP connection can end up choosing different backend servers for the same flow when the selection logic is based on system specific calls like `random()`. Similarly, a rate limiter that restricts the number of maximum sessions for a given client can end up rejecting/terminating different connections due to races in the NF threads accessing a shared connection variable during replay.

FTMB [150] overcomes non-determinism by rigorously tracking and ensuring that all the events that can potentially lead to non-determinism (any shared state access

and outcomes of unpredictable system calls) are captured and committed to the stable log before releasing the packets. In doing so, even the benign access to shared variables or non-deterministic calls whose impact is ephemeral to packets processing (*i.e.,* which do not impact the external view) are logged and enforced at the replay node, which not only accounts to excessive log overheads but also limits the NF's operational throughput. Further, with multi-threaded NFs, during replay it enforces a strict ordering for accesses to any shared resources across multiple processing threads. Enforcing this kind of ordering requires more intricate instrumentation of the NF's code and affects both graceful and recovery mode performance.

12.2.5 Data Center Power Infrastructures

State-of-the-art data center power delivery infrastructure can support multiple power sources, allowing some of the racks in data center to be completely powered by the renewable resources and part of the racks to be powered by the non-renewable based sources [151, 152]. We note that, with such configurations, only part of the rack or nodes in a specific isle can exhibit power outages, while the rest of DC does not.

To accommodate such configurations, the standby (backup) nodes can be maintained within the same data center on the racks powered by non-renewables. In the case of GDC that is fully powered by the renewables, the backups will have to be maintained in another SDC which is powered by brown/non-renewable power resources.

As we intend to support both the kinds of power infrastructure. Therefore, we refer to the VNFs in GDC or the VNFs placed on a node that is powered by renewables as the Transient VNFs and the standby VNFs that are placed on a node powered by non-renewables either in SDC or in the same data center as Stable VNFs or the replica VNFs.

Our main goal is to provide generalized framework for VNF state replication that exploits the opportunities and overcome the challenges (Refer 12.2.1) to deploy the VNFs in GDC and to provide high availability of NFV services through efficient state replication mechanism.

Chapter 13

Related Work

In this Chapter, we present the literature survey on the state-of-the-art work in the following related aspects:

- NF Migration Frameworks,
- HA and FT for NFs and SFCs, and
- NFV related work towards employing Green energy and addressing the energy efficiency prospects for NFV.

Contents

13.1 Resiliency and Fault-Tolerance

13.1.1 Network Function Migration

Split/Merge [72] presents the application level state migration for the middleboxes. It defines state access APIs to read and update the internal state of virtualized NFs being moved across hosts. It relies on the ability to identify per-flow state to provide consistent migration. OpenNF [121], presents a control plane architecture to facilitate loss free state transfer of NF state. SDN controller co-ordinates the

state migration of NFs from one node to the other and relies on explicit events between the source node, controller and destination node to perform the loss-free state transition and to buffer the interim packets at the controller node. However, due to controller based orchestration and event buffering mechanism, it imposes high per packet latency for the migration operation.

In contrast, We seek to build application level NF state migration framework that does not depend on specific state update APIs, instead rely on minimal annotations by NFs that can assist the VNFM to sufficiently track, distinguish and update just the dirtied application state. Further, to optimize on latency, we seek to avoid any controller intervention, but handle the state migration within the context of VNFMs and NFs only.

13.1.2 Fault Tolerance and High Availability

Pico Replication [153] is an application level NF state checkpointing based high availability framework built on top of Split/Merge. It provides fine-grained flow level state replication and employs flow group based NF state transfers. In order to enforce correctness, it buffers all the output packets for the duration of NF state checkpointing, thus delaying outputs even under failure-free operation.

FTMB [150] is a replay based framework that logs all the input packets and per packet access log of all the determinants (*i.e.,* the shared variables in NF that account to non-determinism) that are necessary to restore the states on the replica during replay. In addition, to amortize the cost of input logging, it also employs periodic check-pointing of the NFs. Thus, FTMB guarantees correctness of operation at the replay mode by ensuring strict ordering of packet processing (guided by the packet access logs) at the replay node. In essence, the FTMB notion of correctness emulates strict idempotent per packet behavior across the active and replica nodes. This comes at the cost of accounting multiple per packet access logs, which potentially become the bottleneck for NFs with more than 5 shared variables, for packet rates of 1.25Mpps (refer: section 5 of [150]), resulting in more than nearly 30% overhead traffic. In addition, due to periodic VM snapshotting, the tail latencies drastically increase from less than 100 μseconds,at the 50th%-ile to nearly 810 μseconds, at 95th%-ile and 18ms at 99th%-ile. Also, neither of the work account for chain of network functions. We seek to fill this gap.

13.1.3 Alternative Architectures

StreamNF [154] and StatelessNF [155] present alternative approaches of externalizing the state of the network functions to in-memory databases like RAMCloud [156].

While this may be feasible for some network functions, the database can become a bottleneck, plus substantial refactoring is required. Key advantage with externalized state is that when any NF instance fails, the state in still available for the replica NF to seamlessly failover provided the flows are redirected to the replica NF. It can be noted that the externalized state approaches better suit the Microservice architecture [157], however, there are two fold limitations and performance challenges with externalizing the NF state.

- First, the NF packet processing rate which at line-rate (10-40Gbps) is expected to support 10-15Mpps (million packets per second), gets limited by the read/write transactions supported by the external database which ranges in the order of 4-6Mtps (million transactions per second).
- Second, the need to instrument and refactoring of the NF code to externalize the NF state require all the internal NF state entities to be expressed in well-defined key-value store mode, which can be easily dealt for the per-flow states, but rather intricate to express the shared per-session state and internal state variables.
- Third, typically NFs allocate and release memories dynamically (via alloc and free callback functions as in nDPI), and are usually ephemeral - which would result in excessive overhead.
- Finally, the major challenge arises in ensuring the correctness and consistency of the externalized state w.r.t. the failed NFs that might have partially processed the packets and updated the states, before crashing. In such scenarios, both the NFs and the database need to maintain additional version control for each state updates so that state updates can be validated before commit, which would further reduce the NF processing capacity.

Architecturally and conceptually having externalized state for NFs might as well seem a step in the right direction, but the challenges highlighted above in terms of performance and complexity of operation leave us little reserved on employing them to high speed packet processing network functions.

13.2 Implication on NFV with Green Energy DataCenters

We observe the prospects of employing Green energy for the VNFs is a less studied topic. Nonetheless, we discuss some relevant work in the overlapping categories.

Table 13.1: Comparison of the related state-of-the-art solutions for NF and NF Chain Resiliency.

Work/Feature	Pico Replication	FTMB	Stream NF	Stateless NF
High Level Architecture & Approach				
FT scheme	Application level Check-Pointing	Replay + VM Check-Pointing	Externalize state + Replay	Externalize state <No replay>
Redundancy scheme	Active-Standby (1:N)	Active-Standby (1:1)	Active-standby M:1	Active-standby M:1
Recovery scheme	Only reroute flows by SDN controller; No Replay. state is check pointed.	Replay mode: Replay packets from Input logger + PAL from Output Logger	Replay mode: Replay packets from Root node + play NF in replay mode	Re-route flows to replica instance
Instrument MB code	Minimal	High*	High	Minimal
Failure Detection	SDN Port_down & connection down	Not Addressed	Not Addressed	Not Addressed
Failure Types	Soft (NF) Failure: ✓ Link Failures: ✓ Node Failures: ✓	Soft (NF) Failure: ✓ Link Failures: ✕ Node Failures: ✕	Soft (NF) Failure: ✓ Link Failures: ✓ Node Failures: ✓	Soft (NF) Failure: ✕ Link Failures: ✕ Node Failures: ✕
Checkpoint Freq/s	1000	5-50	–	–
Service chain	NO	NO*	YES	NO
Failure Free Overheads				
Latency Overhead (ppktmedian,99%ile)	15-20% (2-3ms) 5sec?	30us 35ms	2-3us 13ms	65-300us 1ms?
Throughput impact (% drop)	3-5% 50% due to pkt copy	5-30%	Not Reported	∼15-20%*
Measured Packet/Data rate	10-200Mbps 10-200KPPS	1-10Gbps* 1.4-4 Mpps	Not Reported	1-8Gbps ∼4.7Mpps
Service Unavailability	RTT to SDN controller	Not specified	Not specified	Not specified
Restoration Time	Order of milli sec.	40-300 ms	Not specified	Not Applicable

13.2.1 Green Energy and Energy Efficiency

In [158], authors analyze the prospects of energy efficiency that can be achieved by employing the VNFs for the Evolved Packet Core, Customer Premise Equipment, and Radio Access Network in telecommunication networks. This study is seminal in terms of establishing the energy efficiency prospects of employing the VNFs, but does not study or account the consequences such deployment.

In [28], authors present the cloud service provisioning scheme to enhance the stability of smart grid and maximize renewable energy and cloud resource utilization by accounting the variability and uncertainty of both the cloud services and the generation of power by the renewable energy sources.

In [159], the authors present a green abstraction layer (GAL) that provides more sophisticated power management mechanisms for the routers, switches and NFVs, specially in the Software Defined Networks (SDN) based networks. In [160], authors analyze and present the energy efficiency implications of NFV for different packet processing mechanisms. In contrast, we consider a more broader perspective and target towards achieving energy efficient network infrastructure that can be powered by renewable resources and still be able to meet the high availability and resiliency requirements.

Chapter 14

Resiliency Framework: REINFORCE

Contents

14.1 Introduction

Work in the past [150, 153] has tried to address fault tolerance and high availability for individual network functions. Such approaches have excessive overhead when adopted for service chains as we show below. Alternatively, some chain-level approaches [121, 154] seek to provide reliability guarantees across several NFs, but incur very high latency due to packet buffering delays. Designing comprehensive failover mechanisms that can efficiently provide fast failure recovery for single NFs as well as service chains, in either a single node or spanning multiple nodes is the goal of our work. Additionally, we aim to guarantee consistency properties for NF state and packet content under all failure situations. REINFORCE ensures that the external view of the coherent state of an NF or a service chain with its backup is consistent, while achieving external synchrony [161, 162].

We recognize two distinct pieces of information to be maintained for effective failure resiliency in an NFV environment. The application state (state for an NF or chain of NFs) and the packet processing progress (per-flow logical timestamp). When a chain of NFs is backed up on a different node, we use lazy checkpointing of application state to reduce overhead during normal operation and buffer input packets at a predecessor node in between the checkpointing instants. These input packets are replayed to the backup node in case of a failure. Keeping track of packet processing progress of all the flows (with a per-flow timestamp) then becomes the minimum, critical information necessary to enforce correctness when the packets are replayed. The application state (state of NF or chain of NFs) can then be correctly recovered through replay. This allows us to commit a minimum amount of lightweight per-flow timestamp information at a finer timescale, while committing the more heavyweight application state at a coarser timescale.

REINFORCE guarantees correctness and achieves external synchrony by speculatively processing packets and compactly committing per-flow timestamps linked to the checkpoints sent to the standby. We precisely replay to the backup only those input packets that had been processed by the primary between the checkpoint instant and a failure. Thus, unlike FTMB [150], we eliminate the need for per-packet access logs at the NFs and also the need to enforce strict ordering of packets while replaying packets at the backup even in a multi-threaded environment. Unlike Pico Replication [153], REINFORCE hides the replication latency and improves throughput by batching and overlapping multiple commit transactions, while allowing the NFs to continue speculative execution with the judicious use of multiple buffer stages. These improvements result in a dramatic performance improvement over existing approaches. To summarize, our key contributions include:

- **Integrated Resiliency Framework**: We present an efficient NFV resiliency framework for DPDK based network functions and service chains with distinct local and remote redundancy schemes (Section 14.3.1).
- **Light Weight Application checkpointing**: We design mechanisms to minimize the state that needs to be replicated to the backup by taking advantage of *logical clocks*, *external synchrony*, *2-phase commit*, and *dirty state tracking* to enforce correctness before releasing packets from a NF service chain (Section 14.3.2).
- **Chain wide recovery**: We present low overhead and low latency approaches to achieve consistent recovery of all the network functions in a chain within or across hosts (Sections 14.3.2.2-14.3.2.3).
- **Fast Failure detection**: We employ mechanisms to quickly detect NF (in order of μseconds), link and node failures (in milliseconds) (Section 14.3.3).
- **Optimization techniques**: We exploit non-blocking, pipelined NF processing with judicious batching and buffering to maximize the throughput, minimize latency and avoid overhead during normal processing of the network functions (Section 14.3.4).

14.2 Design Considerations

We first present the key requirements for an NF resiliency framework, different NF deployment approaches, means to characterize NF state, and state-of-the-art failure detection and failover solutions. From these, we highlight the key design aspects considered in building the REINFORCE framework.

There are a number of key requirements for building an NF Resiliency framework for service chains:

Correctness and Recovery transparency: NF state must be preserved and consistently recovered across the replica nodes in the event of a failure. In addition, for a chain of NFs it is necessary to ensure that all the NFs in the chain are able to process flows without interruption, by preserving the necessary processing state at each of the NFs in the chain.

Low Overhead: NFs are typically expected to process millions of packets per second and serve large numbers of flows. CPU cycles and memory bandwidth are at a premium. It is necessary to ensure that the performance impact of resiliency support on NFs during normal operation (processing rate and latency) is minimal.

Generality: Given the diversity of types of network services and different deploy-

ment patterns, it is necessary to ensure that resiliency solution can be easily adopted for different types of network functions; it is important resiliency support be incorporated with minimal modifications to the NF's code.

14.2.1 Deployment and State Management

Deployment: Our implementation focuses on NFs run inside containers, although many of our techniques can be generalized to other approaches. Containers enable cheap snapshots using tools like CRIU (Checkpoint Restore In Userspace) [163]. Unfortunately these cannot be trivially applied to NFs because they do not interact cleanly with user-space I/O frameworks like DPDK [45]. Further, they cannot provide consistent checkpoints across groups of NFs run in different containers. For this reason, we develop a resiliency abstraction in the NF framework that can identify just the key NF state that needs to be backed-up in each container.

Service Chaining: NFs are typically chained to efficiently process flows through multiple functional components. For example, we may have a Service Function Chain (SFC) for HTTP traffic to traverse through a NAT, Firewall, IDS, and Load-balancer NFs [16]. The ordering of the NFs needs to be preserved, even when failures cause the flow to be routed to a replica. The NF chain (ordered list) needs to be treated as a unit of processing rather than treating the individual NFs in isolation.

State Characterization: Network Functions keep a variety of state information, including configuration parameters, counters, flow connection status, and application specific variables. We focus on stateful NFs, *e.g.,* NAT, DPI, IDS etc, which may maintain global configuration state, as well as per-flow state. We further classify state updates as either deterministic or non-deterministic, as discussed in the next section. Finally, we also must consider the packets traversing a service chain as state themselves because NFs may modify their data, and we must track their progress through the service chain using logical timestamps. For correctness, all of this state must be properly synchronized to the backup for each NF in a chain.

14.2.2 Failure Model and Detection schemes

Fast failure detection is key to providing fast failover. Here, we only consider Fail-stop software and hardware failures, such as software crashes, link status changes, power outages, *etc.*

Software Failures: For software failures, we rely on low level kernel events like signals, traps and syslogs that can be effectively checked (queried or polled) to

determine the status of individual software NFs. REINFORCE assumes that such failures can be recovered by reloading the NF with a checkpoint of its recent state and reprocessing any intermediate packets.

Link and Server Failures: For hardware failure detection we considered different state-of-the art Layer-2/Layer-3 schemes such as LACP and OSPF, including the SDN and Openflow based Echo and Fast Failover schemes. Ultimately, we selected Bidirectional Forwarding Detection (BFD) [164–166], which is a lightweight, protocol-independent liveness detection protocol that can detect link failures in millisecond timescales. By examining the status of multiple links we can also use BFD to detect server failures. BFD does not take any remedial action, but simply triggers event driven notification, which enables the NF Managers to quickly react to failures, and as a layer 3 failure detection protocol, BFD is well supported by the hardware on routers with ASIC-based forwarding planes.

14.2.3 Recovery: Replay vs. No-replay

Pico Replication [153] first proposed NF resiliency with a pure checkpointing (i.e., No-replay) scheme that buffers all the output and halts NF processing until the completion of the checkpointing (to assure state consistency). However, this buffering results in high latency and degraded throughput during normal operation.

An alternative proposed in FTMB [150] is to maintain input packet logs (at a predecessor node) and replay the log to reconstruct lost state after a failure. With this approach output packets can be preemptively released before creating a full checkpoint, since state can be recreated via replay. This overcomes the latency impact for a majority of the packets, but adds complexity to NF development and can incur high overhead to enforce a sequential ordering.

REINFORCE uses a combination of infrequent (lazy) checkpointing and replay of packets. The key is to maintain external synchrony [161]: rather than provide strict synchronization where NFs block until replication completes, REINFORCE allows NFs to continue speculative execution of packets while a backup is performed. Packets will only be released from an NF service chain once the backup has all the information necessary for recovering from a failure. Relaxing from synchronous replication to external synchrony means that processing can continue through the service chain and for subsequent packets in the flow while still providing consistency guarantees to clients receiving the packets. When a failure occurs, the backup node can replay packets that have to be processed since the last checkpoint snapshot and update the NF application state on the backup. A logical timestamp is used to determine the packets that have been released since the checkpoint, so that the

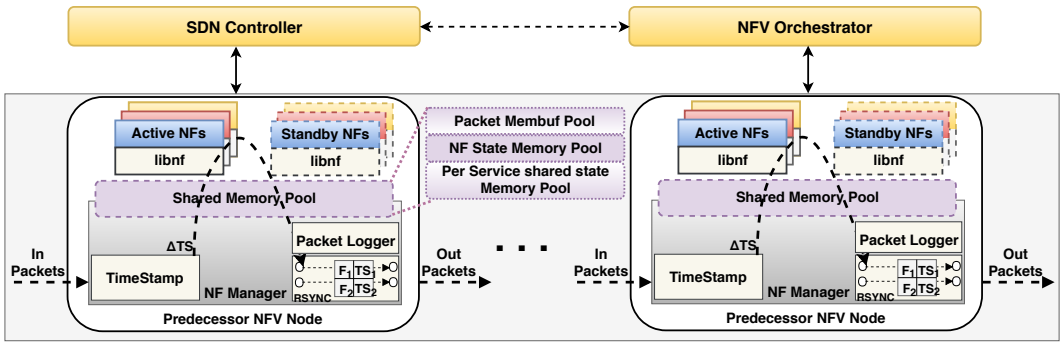


Figure 14.1: Architecture of REINFORCE

replay process does not transmit unnecessary duplicate packets downstream while updating the backup state.

14.2.4 Non-Determinism

We acknowledge the existence of non-determinism, and present an alternative, simpler approach to tackle it without the need for per packet access logs or enforcing the strict ordering of packet access to shared variables.

We exploit the fact that non-deterministic updates are typically tied to specific packets, e.g., the first packet in a flow may cause updates at several NFs, while subsequent packets do not. Deterministic packets can be replayed with no problem. When an NF performs a non-deterministic state update (which we require the programmer to annotate) we link it to the packet which triggered it. Then, due to our use of external synchrony, we only need to ensure that by the time the packet reaches the end of the service chain and is ready to be sent out, all of its dependent non-deterministic state has been checkpointed to the standby, avoiding the need to replay it after a failure. For example, in the same load balancer example, it is sufficient to track the initial connection state update at the start of the flow, rather than tracking and enforcing the access to shared global counters for every packet processed by load balancer NF threads.

14.3 Architecture and Design

We present the key components of REINFORCE and briefly describe their roles. We then present the design of REINFORCE in 3 subsections: failure detection; redundancy schemes (for both a single NF and service chain) and failover mechanism, whether on the same (local) host or on a remote host; and briefly describe

optimizations that achieve high performance and correct failover.

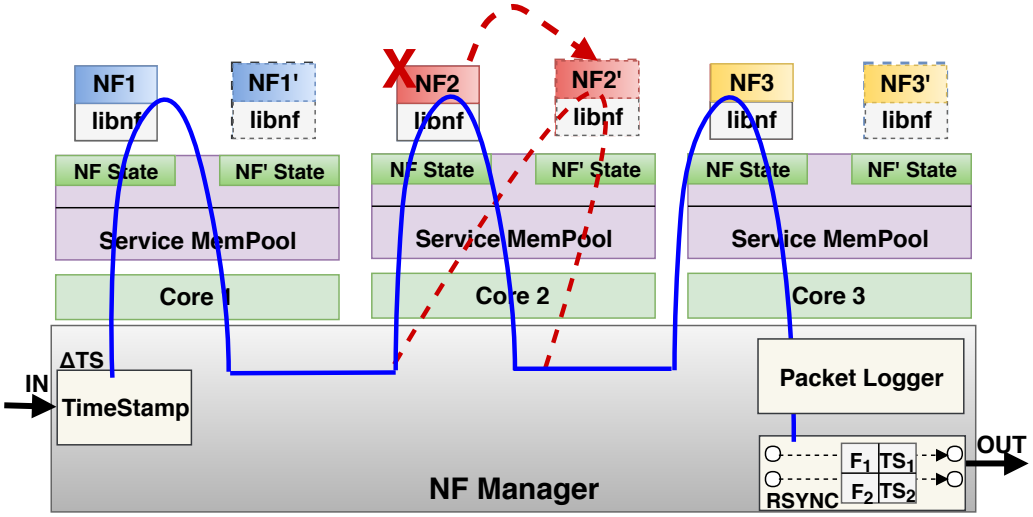


Figure 14.2: Local NF Instance Failover: On an NF instance failure, REINFORCE migrates processing to a local standby (replica) NF.

14.3.1 REINFORCE Components

The key components in the REINFORCE framework are shown in Figure 15.1.

NF Orchestrator is responsible for provisioning the NF Manager nodes and designating the active and standby nodes for different service chains. It also configures the BFD settings on each of the NF Managers in the cluster.

SDN Controller is responsible for populating the flow entries and forwarding rules at each of the NF Manager nodes. In addition, it pro-actively configures the back-up path options: a) in the case of multiple links, it configures the alternate output ports on the predecessor nodes of the designated active NF manager node; and b) configures the flow rules on designated replica standby nodes.

NF Manager is the core component of REINFORCE. It acts as the overall in-host controller of the NF functionality, depending on DPDK's framework for zero-copy delivery of packet data to and between NFs of a service chain within the host. As shown in Fig. 15.1, in addition to providing packet-switching within the host, the manager is responsible to track the liveness of associated network ports (links) and the NFs provisioned on it. It also provisions and provides the shared memory pools to

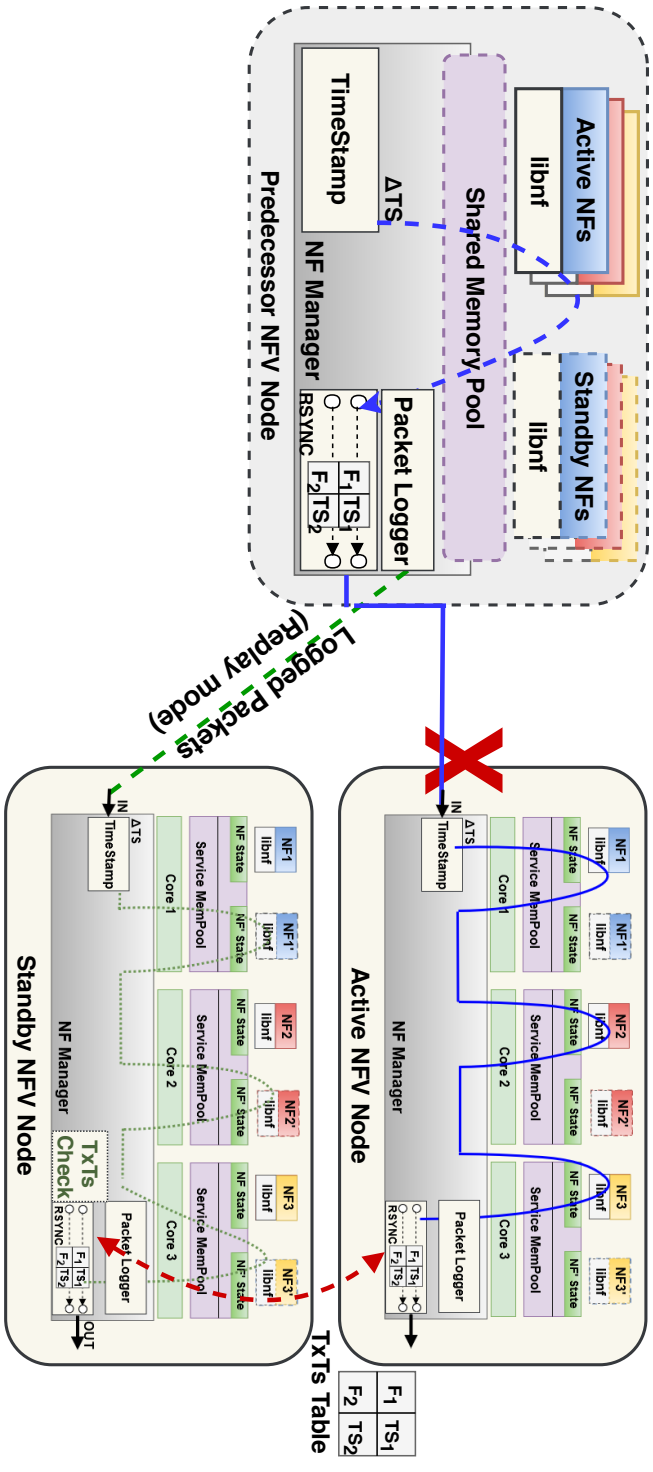


Figure 14.3: Remote NF Chain Failover: On Link or Node failures, the neighbor node in REINFORCE initiates failover to a remote standby (replica) node.

the NFs to exchange packets, shared memory state, and message notifications. Each NF that is a candidate to be protected against failures is integrated with a "*libnf* " library that provides the necessary hooks, thus minimizing the changes required on the NF. In addition, it implements the "packet logger" module to log and timestamp all the incoming packets, and "RSync" module to provide consistent state replication service to the NFs. We leverage both proactive and reactive configuration schemes along the lines of [15, 18]. For software failures of NF instances, the NF manager ensures fast and transparent failover to a local hot-standby NF. To address link or node failures, the NFV Orchestrator, SDN controller and NF Managers coordinate together to provide chain wide failover.

14.3.2 Resiliency framework

In this section, We describe our resiliency framework in terms of the applied redundancy scheme, local and remote replication mechanisms to account efficient failover for different failures. Our failover mechanism is built to accommodate the desired hierarchical levels of resiliency for distinct service chains to suit the ETSI defined resiliency levels.

14.3.2.1 Redundancy Scheme

We make use of the "Active–Hot Standby" configuration to facilitate NF resiliency, where the state updates from the active NFs are consistently updated on the corresponding standby NFs. As memory is shared between the NF manager and NFs, the manager takes care of replicating NF state while standby NFs themselves are set to sleep state and do not consume any CPU cycles.

REINFORCE supports a variety of failure scenarios at both NF instance level as well as service chain level resiliency. Software failures that can be recovered by NF instances within the same host can be provisioned for 1:1 redundancy of active:standby NFs. When a service chain is protected by a corresponding chain of NFs on the same host, we protect each individual NF instance of the chain, thus allowing REINFORCE to be resilient to multiple NF failures on the same node.

Second, we support failures at the chain level, where all the network function instances of a chain are provisioned on a remote node (which can as well host other active NFs) in standby mode. This supports link and node failures, whether hardware or software failures. Multiple NF chains on different active nodes can be configured to share the same standby node. Thus, we support M:N redundancy, enabling the sharing of a single standby node by several active chains from other nodes. This allows us to limit the number of required standby nodes.

For a local standby, we spend additional CPU cycles for the active NF to synchronize the NF state on the standby NF. This, however is transparent to the NF and is completely handled by a set of library functions, “libnf". For a remote standby, the NF Managers on the two nodes have an “RSync" thread dedicated to handle the atomic state updates between them. Our evaluation shows that remote replication consumes relatively little CPU and network bandwidth.

14.3.2.2 NF Resiliency with Local replicas

In scenarios where only software crashes must be tolerated, the replica NF (also termed standby or backup) is provisioned locally in the NF Manager to provide resiliency from NF instance failures as shown in Figure 14.2.

Standby NF: The NF Manager arbitrates the wake up of the standby NF. After initialization, the backup remains in Pause state until the NF Manager releases signals to wake up the NF. Once the NF Manager detects the failure of an active NF, it wakes up the standby NF and lets it process subsequent packets. Thereafter, once the active NF is restored, the standby NF is moved back to the pause state, allowing the active NF to continue processing packets. The NF Manager (along with *libnf* library) brings the NF state up to date before letting the NF start processing the packets.

NF state checkpointing: We use a “no-replay" scheme to synchronize the active and standby NFs when they are on the same host node (local replica). We strictly enforce an “output commit" property: *i.e.,* no output (packet/s) are released by an NF until all of the corresponding NF state is updated on the backup/replica instance. It is achieved by copying the NF application state modifications (state affected by packet processing) to the replica NF's shared memory. CPU overhead and latency for local memory copies varies based on the size of data copied and the number of copy operations required to synchronize the state. To ensure the memory state is consistent when the copy is performed, we trigger checkpoints after an NF finishes processing a packet or batch of packets.

14.3.2.3 Chain wide Resiliency with Remote Replica

We employ both checkpointing and packet “replay" to provide resiliency from host node failures and link failures (that result in loss of connectivity) when the backup is on another node, as shown in Figure 14.3.

Standby Server: The NF Orchestrator designates the standby NF server node and notifies the NF Manager at the node with the Active NFs of the chain as well as

the predecessor node serving the NF chain, as shown in Figure 14.3. The node with the Active NFs and the predecessor node monitor the liveness status using BFD, which we discuss in detail in Section 14.3.3 If an alternate route to the primary server exists after a link failure (*i.e.,* an alternate output port has been configured by SDN controller), the predecessor node simply redirects the traffic. If a link or node failure makes the primary unreachable, the predecessor node initiates the replay mode on the designated standby.

Chain wide state checkpointing: REINFORCE relies on five key concepts namely i) Input logging with timestamps, ii) Latch buffers for external synchrony, iii) Pipelined replication, iv) Atomic state updates, and v) Replay-based recovery to assure consistent and efficient failure resiliency of chains replicated to a secondary host.

1. **Input packet logging with Logical Time stamping**: In REINFORCE, all the packets are logged (buffered) at the input (predecessor node in a multi-host chain) and appended with a logical timestamp (*e.g.,* simple 64 bit packet counter)[40]. The input packet log at the predecessor node is used to replay packets to the standby node when an active node fails. At the active NFV node, the timestamped value of each packet is used to track the packet processing progress for each flow. This information is maintained in a Transmit Timestamp (TxTs) table replicated across the primary and backup nodes. The input logger clears buffered packets upon notification by the active node's NF Manager.

2. **Transmit Latch Buffers** In order to provide external synchrony, packets must be buffered until any state related to packets with non-deterministic processing has been replicated to the backup. If packets did not cause any non-deterministic state updates (which is often the case), then they can be released more quickly since the standby must only record their timestamp in order to know which packets must be replayed in the event of a failure. Packets are stored in a latch buffer at the end of the service chain on the primary server. Once a TxTS table commit acknowledgment arrives from the standby indicating the timestamps and any non-deterministic state has been recorded, the packets are released to downstream external nodes. We implement several optimizations described in Section 14.3.4 to reduce the cost of remote communication and the latency for packets that may have to wait for the state update to the replica.

3. **Pipelined Replication**: Our remote replication scheme simplifies consistency and improves performance by leveraging the local replicas that we already provide

[40] A single nondecreasing counter is sufficient on the Logger; this in turn gives monotonic per-flow counters when packets are demultiplexed on the primary node

for software failures on the primary host. The local replicas have their state updated at the end of each batch of packets, as described above, which gives a consistent version of the state which can be copied to the remote server without any need to pause the primary replica. As discussed previously, REINFORCE differentiates between deterministic and non-deterministic updates to either NF state or packet data. Deterministic updates can be recovered via replay on the remote host, so state checkpoints do not necessarily need to be replicated to the backup for every batch. Instead, each NF in the chain replicates in a lazy fashion to reduce overhead. On the other hand non-deterministic state updates cannot be replayed, so packet batches with non-determinism need to have a checkpoint replicated to the backup before they are released from the primary. Fortunately, this replication can be parallelized with further packet processing in the remainder of the chain until the packet is ready to release in the latch buffer.

4. **Atomic State Updates**: REINFORCE follows a 2-phase commit protocol to provide atomicity between state updates at the backup and packet releases at the primary. Our commit protocol begins when the primary sends its updated Transmit Timestamp counters and any non-deterministic state updates that are necessary. The secondary associates the logical clocks (flow-specific Transmit Timestamps) with the arriving checkpoint state, and ensures that both of these are fully received by all NFs in the chain before acknowledging back to the primary. At this point, the primary can release its Latch Buffer so packets can continue on to their destination. It then notifies the secondary so that it can commit the checkpoint state. State updates resulting from deterministic operations are transmitted periodically; once this state has been received, the predecessor node can be notified to clear its input log.

5. **Replay:** The use of latch buffers and atomic state updates guarantees "external synchrony," i.e., the state maintained at the backup can be made consistent with the output packets that have been released from the primary server. Note that since deterministic application-level state is only replicated periodically, it is possible for the standby to recover to a state where the TxTs table says that some packets have been released, but the backup state does not yet reflect the deterministic updates they should have produced. Thus, in the event of failure, the standby NF chain must rollback to the last checkpoint and replay any subsequent packets so that the external view of the system (outside of the chain) is the same whether the failure occurred or not. However, since a chain has multiple NFs and their state updates may arrive at different times, it is possible for a packet to need to be replayed through some NFs which have already processed it. We believe that NFs are already designed to be robust to receiving duplicate packets—duplicate transmissions are a regular occurrence in wide-area networks, and thus this does

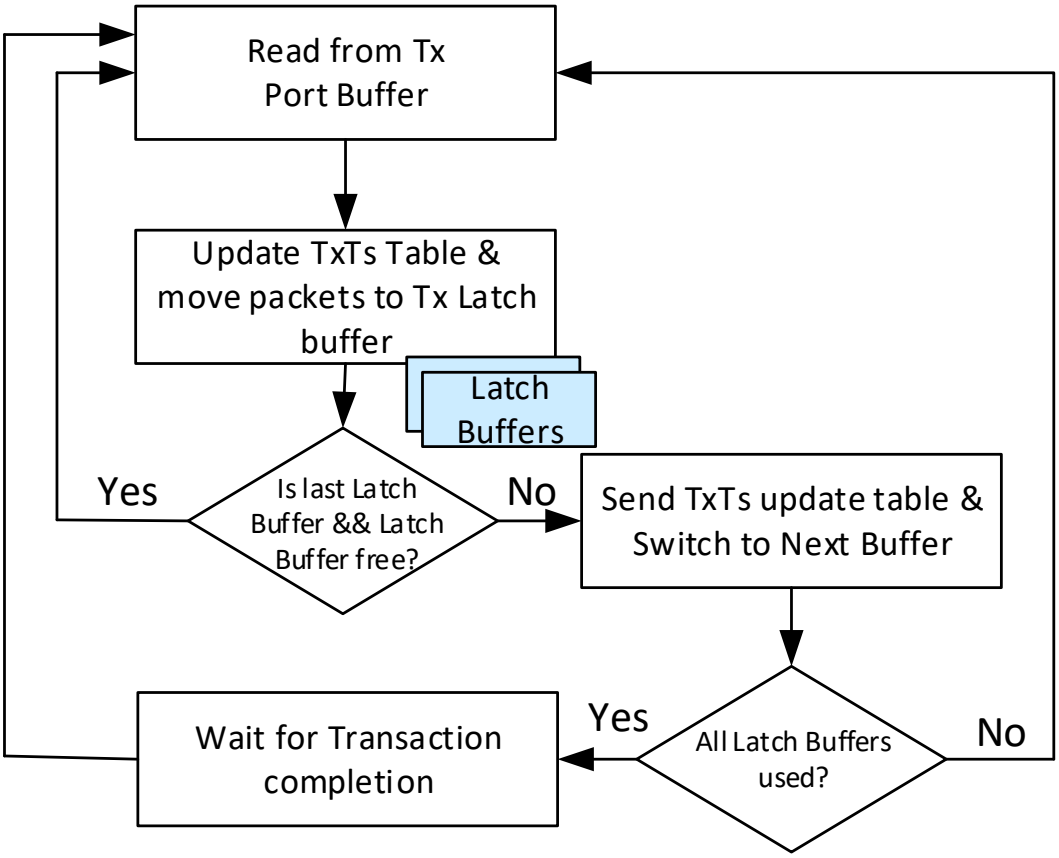


Figure 14.4: Flow diagram illustrating the usage of Multi-transaction Buffers and Opportunistic Buffering.

not require special handling. The exception to this is processing packets involving non-determinism, which is why we ensure tight state consistency for them—such packets are only released once their state has been confirmed by the backup.

14.3.3 Failure Detection

14.3.3.1 NF instance Failure Detection

NF Managers are responsible to track the liveness of all provisioned NF instances. The NF manager detects NF Instance failures in two ways. First, it captures 'voluntary' NF instance failures, by registering for event notification and messages that are triggered via OS (Linux) signals and NF Instance-specific messages, when any catch-

able exception occurs at an NF Instance. Second, for involuntary NF terminations, the NF manager performs periodic (every 100 μseconds) checks via the `kill(0)` signal to check and deduce the status of all the registered active NFs. This operation is carried out by the NF Manager's "Monitor thread" which is also responsible for other tasks such as NF registration, de-registration and logging of statistics. The 100 μseconds probe interval is system configurable. Even at a 100 μseconds probe interval, the CPU overhead is less than 1%, to track the liveness of 64 NFs.

14.3.3.2 Link and Node Failure Detection

We make use of BFD and adapt the configuration settings to mirror those suggested for S-BFD [167] to provide both link and host failure detection.

BFD configuration and Tunable parameters: At the time of node initialization, the NF Manager is configured to initiate BFD in active mode for each of the host's ports. We configure the BFD minimum Rx and Tx transmit timer intervals to 1 millisecond and the detection timeout multiplier to 3. When the session is initiated in active mode, it probes at a low frequency, once every 100 milliseconds (10Hz) for the remote end to establish the connection. Once the session is established, the connectivity probe frequency increases to 1000Hz, so that link or remote node failures are detected within 5 milli seconds[41]. We tuned these values (parameters set at compile time) for our nodes and are more aggressive than recommended for normal network BFD operation [164].

Even in the worst case, with the link operating in active mode and with a frequency of 1000Hz, BFD packets (60 bytes) account to less than .005% of the 10Gbps link bandwidth. Setting the probe frequency $>$ 1000Hz for directly connected links resulted in considerable false positives even at lower traffic rates.

Distinguishing link vs. node failures: When BFD is configured per link, and when nodes share multiple links, it is possible to differentiate link failures (even when another link status becomes active in the next iteration cycle), from node failures (where all BFD links show they are down). This differentiation is necessary to ensure quick failover at the link layer (re-route to a different port) that can be handled locally with no overhead, rather than initiating a full NF chain failover to a designated remote-node.

[41] 3 milli seconds for asserting three consecutive probe failures, plus an additional cycle to confirm if single or multiple links have failed

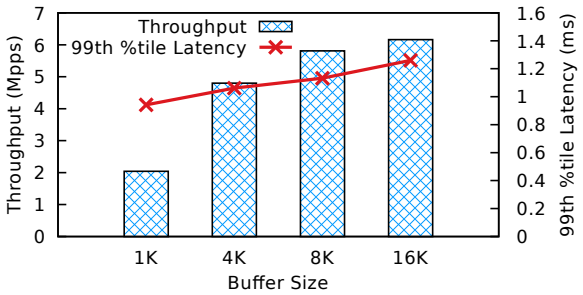


Figure 14.5: Effect of Tx Hold ring buffer size on Throughput and latency

14.3.4 Tuning, Assumptions, Limitations

There are a number of optimizations we used to lower the processing overhead in normal operation. To amortize per packet processing cost, we perform the state replication and packet release tasks across a batch (32) of packets. We also tune the size of the Latch ring buffers to hold outgoing packets. Fig. 14.5 examines the impact on round-trip latency as the size of the output ring buffer varies, for simple forwarding. With a 200μseconds RTT and the input at 10 Gbps, increasing the buffer size from 1K to 4K and 8K can double or triple throughput, while incurring less than a 20% increase on tail latency. By having multiple latch buffers, we achieve concurrency between the replication of state and packet processing.

During replay mode, upstream NFs of an NF chain may process duplicate packets for various reasons. We assume (safely) NFs are able to handle duplicate packets without impacting correctness. Similarly, because REINFORCE depends on logical packet timestamps, we can tolerate packet re-ordering. However, we need to check the timestamping for replay mode to avoid re-release duplicate packets. To accommodate timers, NFs must explicitly annotate them so that the remote standby can instantiate timer events after a failure. We describe these issues in greater detail in [35].

14.4 Implementation

REINFORCE is a capability built onto OpenNetVM [46], which is a DPDK [45] based NFV platform. We implemented the following modules i) Packet logging: to add a logical timestamp and log all the input packets to stable store, which are then recycled after the configured refresh interval; ii) RSync: to enforce external synchrony and perform the two-phase commit transaction using multiple latch

buffers; and iii) Liveness Monitoring: to monitor the liveness of locally provisioned NFs and BFD sessions across the configured links. We dedicate 1 CPU core each for the RSync and Liveness monitoring functions.
Our extensions to the core NF Manager [46] framework are minimal. We extended the control framework to coordinate failover via the pause and resume event notifications to the Active and Standby NFs and also to perform failover actions in case of remote link failures. We modify the default packet-out functionality to conditionally transmit packets to the RSync output ports instead of DPDK NIC ports, to account for the transmit state timestamp for each flow and to enforce 2 phase transaction commit.

14.4.1 Local Failover

Shadow Rings: In a typical NF platform implementation, subsequent to the NF processing, packets are handed to a transmit ring to be sent out on the network link or forwarded to another NF in a service chain. REINFORCE introduces the concept of a “Shadow Ring" on both receive (Rx) and transmit (Tx) ends of the NF processing pipeline. Shadow rings are shared ring buffers between active and standby (replica) NFs. Rx shadow rings buffer the batch of packets that the NF needs to process and Tx shadow rings buffer the batch of post-processed packets for which the state updates have not yet been reflected on the replica NF. They enable enforcement of “output commit" for state update on local standby NF and also minimize packet loss in the face of NF failover. Benefits of a shadow ring are two-fold. First, in normal processing, on the transmit end, they enforce output commit and ensure correctness of the state update on the replica, by allowing the transmission of a batch of processed packets only after the NFs state is updated on the replica. Second, when a local failover is required due to NF failure, the receive end shadow buffer enables the replica NF to immediately pick up from the first unprocessed packet, while allowing to discard the previously processed packets for whom the state-update has already been completed.

Shared Memory pool: Network functions have two kinds of application state, i) External or the shared state across all the NF instances; and ii) Internal state including the per-flow state and instance specific configurations for each NF. To account for shared state, NF Manger allocates and maintains a per service type memory pool. The size of this memory pool is a configurable parameter. In our experiments, we set this to 4 MB (our most complex NF, based on the nDPI library uses 2.8MB). To account for internal NF specific state, the NF manager allocates and reserves a dedicated memory pool for maintaining this state for each instantiated NF instance. The size of this memory pool is again a configurable parameter. In

our experiments, we set this to 64 KBytes.

Tracking Dirty State : To ease development, we do not require explicit APIs for NFs to interact with their state, as is done in prior work. Instead, we define memory pools for each NF state type, but allow NFs to perform arbitrary operations to these regions. REINFORCE automatically detects dirty state regions by scanning small chunks of the NF and service state memory pools to detect changes, similar to FaRM [168]. In our evaluation we configure the number of state chunks to 64, allowing the minimum transferable chunk size for NF specific state to 1KB and for the shared service state to 64KB. The only API that NFs must use when manipulating state is setting a per-packet flag that indicates if it caused non-deterministic updates.

14.4.2 Remote Failover

Atomic Two phase commit transaction: We use a simple UDP like best-effort connectionless transport (while it is desirable to have a reliable transport, we wish to avoid the connection setup and initial handshake overheads of TCP) to deliver updates to the backup and use sequence numbers to identify any missing packets. We use a custom Ethernet type to differentiate state update packets from the regular NF destined packets associated with the DPDK port. If packets are lost, we abort the transaction and resend new updates. State transfer packet headers include the fields to indicate the type of packet transfered (either state transfer or acknowledgement packet), type of state (NF state, service configuration information, or Tx Timestamp), size of the packet, base offset address, packet sequence number and 'last packet' flags. In our testbed implementation, for expediency in our experiments, we have dispensed with the final acknowledgment from the primary to the standby as the last phase of the 2-phase commit protocol, but we expect the performance results to be accurate.

Accounting for failed transactions: When the Tx state update commit acknowledgement is not delivered to the primary, the NF manager may be blocked, resulting in port buffers getting full and subsequent processing by NFs being discarded or stalled. To avoid this, we have a transaction timeout after which the NF manager aborts the transaction and continues to process the subsequent packets and resend new updates.

Tx timestamp state update overhead: We opportunistically perform the transmit timestampe table state updates as often as possible. The frequency of operation is limited by the RTT and number of configured latch buffers on the system. Assuming a best case RTT (across two directly connected nodes) of 100μ seconds,

performing each Tx timestamp checkpoint in the worst case needs to transfer the entire TxTs table of 64KB (64 1KB packets), an overhead of less than 4.25%. Our use of large latch buffers (8K) enables to checkpointing at slow rate (roughly once every 5 RTTs) *i.e.,* performing checkpoints once every 500μ seconds, reducing the link overhead to less than 0.85%.

Sample NFs: We implemented or ported existing NFs to use REINFORCE's state memory pools. NFs such as the Monitor, VLAN Tagger (*e.g.,* for differentiated QoS treatment), Load Balancer, and DPI (using nDPI [169] library) NFs also use the service memory pool to share common state across instances.

14.5 Evaluation

Our experimental testbed has three Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz servers, each with 157GB RAM, running Ubuntu SMP Linux kernel 3.19.0-39-lowlatency. For these experiments, nodes were connected back-to-back with dual-port 10Gbps DPDK compatible NICs to avoid any switch overheads. We make use of DPDK based high speed traffic generators, Moongen [97] to generate line rate traffic consisting of UDP and TCP packets, and wrk [170] to flood HTTP download requests. We vary the number of flows and the NF chain setup as needed for each experiment.

14.5.1 Operational Correctness/ Performance

First, we demonstrate operational correctness of REINFORCE with Graybox and Blackbox tests.

14.5.1.1 Graybox tests

We first validate correctness of failover operation through instrumented template NFs that check for consistency of NF state updates and packet processing. If any packets are obtained with incorrect content (inconsistency between state embedded in the NF and packet content) then the NF flags the error to the NF Manager and the NF terminates. We run a script to perform 10K forced terminations and re-activation of the Active NF. Each time the Active NF is forced to fail, the failover to the backup NF happens automatically. and when the active NF is re-instantiated the NF state is updated and flows are re-routed back to the active NF.

14.5.1.2 Blackbox tests

We assess the application level of failover through end-to-end test scenarios.

DPI based protocol detection: We demonstrate REINFORCE with a DPI NF and feed it PCAP traces available at NTOP [169,171]. We observe that the DPI NF identifies the protocols correctly both without failures and when the primary fails and REINFORCE fails over to the backup.

Table 14.1: Using Pcap traces to verify correctness

Pcap_trace	Detected Protocols
mpeg	MPEG
Hangout	GoogleHangout
Youtubeupload	YouTubeUpload
Snapchat	SSL_No_Cert, Snapchat
QUIC	GMail, YouTube, Google, QUIC

Table 14.2: Effect of Failure on HTTP downloads

	Baseline	w/o Res	w/ Res
Total requests	652	294	626
Total read bytes	65.84GB	31.14GB	63.83GB
Requests/sec	10.85	4.9	10.43
Transfer/sec	1.10GB	531.07MB	1.06GB

HTTP downloads: We route the HTTP downloads through a service chain of 2 NFs (VLAN Tagger and Monitor). We initiate repeated HTTP download requests for a period of 60 seconds, and trigger the VLAN Tagger NF instance failure at the 30 second mark. We observe the impact on the HTTP downloads without REINFORCE (w/o Res) and with REINFORCE's failover mechanisms (w/Res) and compare to baseline failure-free case. There is very little impact on the application (Table 14.2 and Fig. 14.6). There is a dramatic improvement with REINFORCE after the NF's failure at 30 secs. In all these experiments, we configured the number of latch buffers (*i.e.,* multiple transaction buffers) to 3.

Failover Times: We measured the times for local and remote failovers from the

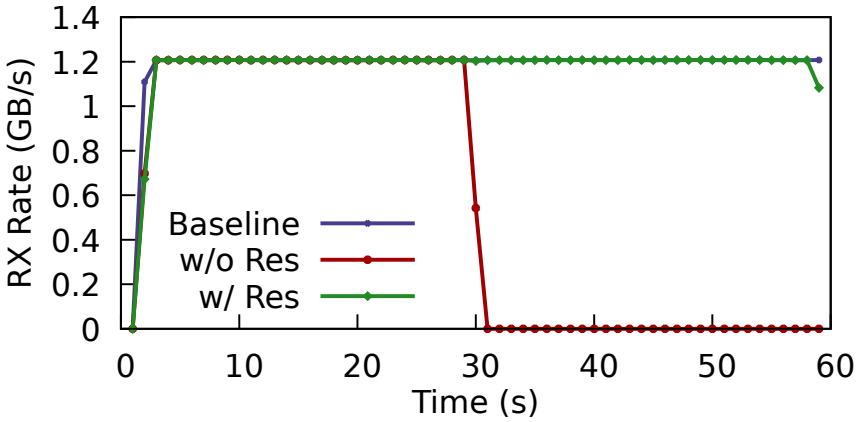


Figure 14.6: REINFORCE has minimal effect on HTTP downloads compared to the the baseline failure case

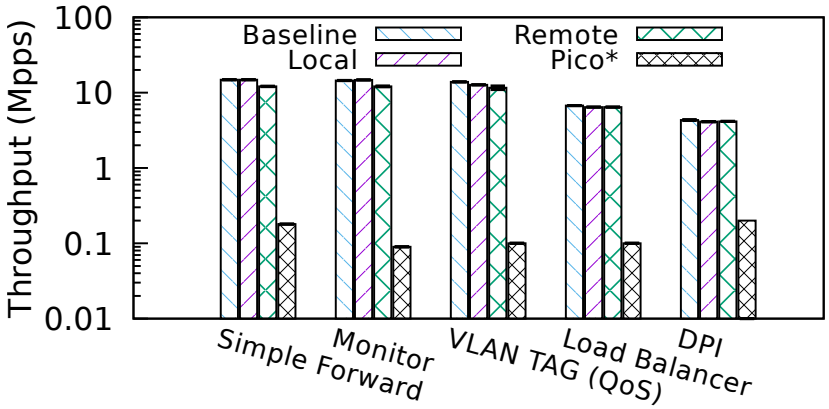


Figure 14.7: Effect of local and remote Replication on normal operation for different NFs.

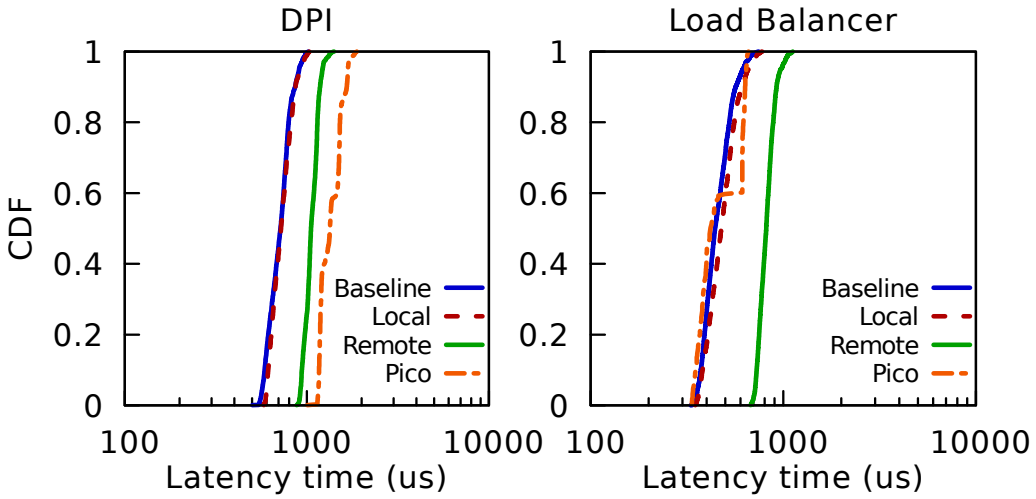


Figure 14.8: CDF of packet latencies for DPI and Load Balancer NF Instances with different replication schemes.

instant we induce a failure. For local failover: mean= 56μs and maximum= 114μs. For remote failover: mean= 5917μs and maximum= 6441μs. This includes failure detection time with BFD and for the predecessor node to initiate the failover at the backup by starting replay of buffered packets.

14.5.2 REINFORCE vs Pico Replication

We compare REINFORCE with Pico Replication [153][42] for a number of different NFs in terms of i) overhead during normal operation reflected in throughput, and ii) latency of packet processing (additional state update operation), for individual NF instances. Fig. 14.7 shows the normal operation's throughput, in Mpps (note log scale, along with error bars). Local replication performs almost as well as baseline (no resiliency). REINFORCE's remote replication sees slightly lower throughput, but still achieves near line rate ~12.5Mpps throughput for NFs like Simple Forwarder, Monitor and VLAN Tagger (QoS). More importantly remote replication still outperforms Pico replication by 2 orders of magnitude. Fig. 14.8 shows the impact on packet latency for different NFs. Local replication adds less than 5 μseconds to the baseline case. Remote replication adds roughly 400 μseconds (we show results for two selected NFs).

[42] We implemented the application checkpointing and output commit policy as presented in [153].

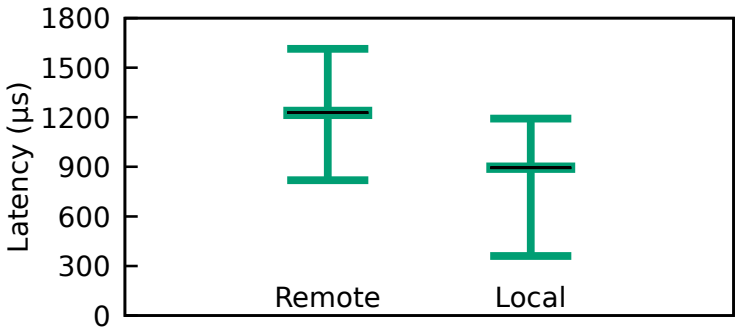


Figure 14.9: Measure of latency for flows configured with different resiliency levels

14.5.3 Differentiating Resiliency Levels

We demonstrate the benefit of REINFORCE's ability to have different flows configured with different resiliency levels. REINFORCE provides the desired resiliency while isolating them from each other. We have a single Monitor NF, configured to: (a) one NF instance with only local resiliency (backup on the same node for flow-1); and (b) another NF instance with resiliency at the node-level (remote standby) for flow-2 and chain2 with only local replication for the Monitor NF and input two flows, one for each chain. Observe the difference in the latency for the two flows in Fig. 14.9, because REINFORCE provides different levels of resiliency for the two flows.

14.5.4 Impact of Chain Length

In this experiment, we consider two experiments with different chains lengths. First, Chain1 has 2 NFs: QoS and Monitor. In the second experiment, Chain2 has 3 NFs: QoS, Monitor and Load balancer. We feed line rate (10 Gbps, 64 byte packets) input and measure the overhead of state replication to the backup during normal operation, in terms of impact on throughput and latency. We use 3 buffers. We compare four cases: baseline (OpenNetVM with no resiliency); Only local replication; 'full' REINFORCE with a remote replica; and finally a comparison with Pico Replication. We show the throughput (in Mpps, note the log scale on the Y-axis) in Fig. 14.10. Going from chain length 2 to 3, the baseline and local replication see close to 10 Mpps with very little impact. Remote replication sees a small reduction in throughput. However, REINFORCE has a throughput that is 100x better than Pico Replication.

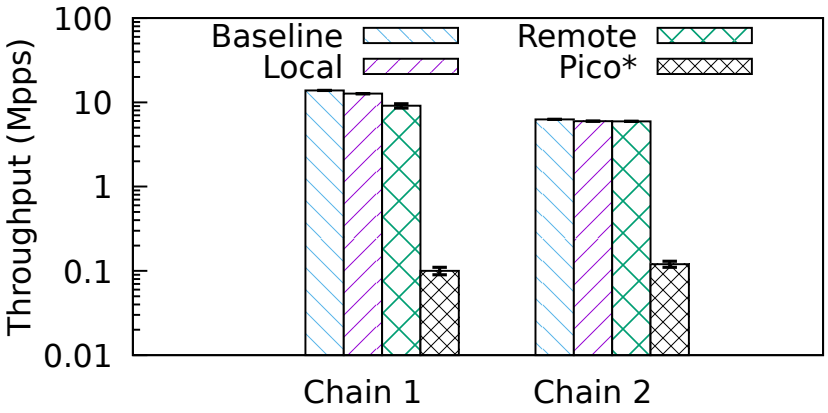


Figure 14.10: Performance impact on chain processing due to local and remote replications.

14.6 Conclusion

To our knowledge, REINFORCE is the first work to address chain wide network function resiliency, supporting detection and recovery of software, server, and link failures. REINFORCE automatically tracks and replicates state to standby NFs while enforcing correctness.

Local replication of NFs on the same host is extremely lightweight—incurring less than 1% overhead—by exploiting the shared memory framework of OpenNetVM. The amount of state replicated is minimized by using 'lazy' replication of NF application state across hosts, and packet replay is used to speed up recovery of deterministic NF processing.

Non-determinism in NF processing is also handled by requiring programmers to annotate such updates, and avoiding packet replay at the standby after a failure. We introduce a logical timestamping scheme that tracks packet processing progress and exploit external synchrony to allow pipelined processing and replication across a chain. Performance results show that the overhead of REINFORCE yields a 100x throughput improvement over Pico replication, a current state of the art technique.

Chapter 15

REARM: Fueling the Green Energy Data Centers

Contents

15.1 Introduction

As outlined in Section §12.1.3, the renewable energy backed DCs or the GDCs are vulnerable to disaster due to power outage or power shortages. To account for *Disaster Recovery* (DR) in such circumstances, we present our work REARM[43].

REARM aims to enable running the VNFs in renewable energy backed data centers while providing sufficient degree of reliability and high availability. REARM makes use of the most preferred active-standby redundancy mechanism with many-

[43]REARM: **R**enewable **E**nergy b**A**sed **R**esilient deploy**M**ent of Virtual Network Functions.

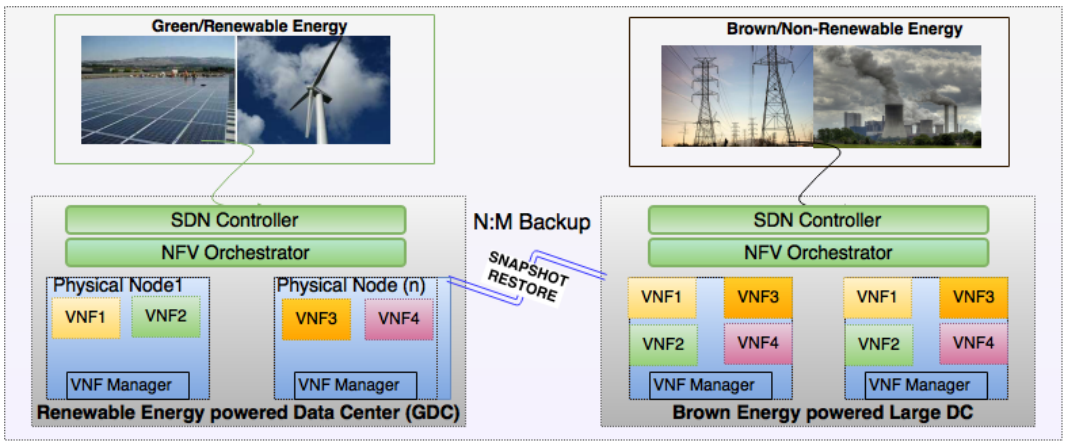


Figure 15.1: REARM Architecture.

to-many[44]backup model [150,172] to address the sustained VNF availability even in the unstable and intermittent power outage issues faced by GDCs.

The key contributions of our work include:

- Distinction of NFV heterogeneity, different deployment models and the associated challenges in providing resiliency and high availability.
- Introduce REARM, that presents a generalized NFV framework to migrate VNFs efficiently and seamlessly in order to achieve high availability and resiliency to power instabilities of GDCs.
- Design and implementation of novel push-pull and adaptive threshold based state migration techniques, that enables to lower the warning times on the order of milliseconds.
- Preliminary evaluations on a prototype testbed using the controlled experiments to demonstrate the benefits of employing REARM.

[44]Multiple NF instances of same type in GDC can be backed-up by one or more NF instance in SDC.

15.2 REARM Architecture and Design

15.2.1 REARM: Architecture

We envision SDN based network with NFV management and Orchestrator to provision the VNFs on physical nodes. REARM's architecture is based on the ETSI-NFV framework [15]. Figure 15.1 shows the high level architecture and key components of REARM. We briefly describe the role of the key components and concepts of REARM's architecture.

15.2.1.1 NFV Orchestrator

NFV Orchestrator (NFVO) is responsible to manage, coordinate and communicate with the VNF managers to handle snapshot and restoration of stateful VNFs in a data center.

15.2.1.2 VNF Managers

NF Manager is responsible to instantiate the VNFs and to perform snapshot and restoration of VNFs on a physical node.

15.2.1.3 SDN Controller

SDN Controller is responsible to implement the flow policies, *i.e.,* to configure the service chain for the flows and to setup the forwarding rules/route for the flows to traverse through the desired chain of VNFs within the data center. Also, the SDN controller is responsible to migrate and re-route the flows to the appropriate VNFs in the data center.

15.2.1.4 Advance Warning Time

We leverage the YANK [29] concept of advance warning time (AWT) to backup the transient VNFs on the stable nodes. AWT refers to the estimated period until when the renewable energy backed universal power supply (UPS) is expected to last and sustain running the nodes reliably. The computation of AWT depends on the UPS capacity and the estimated power consumption of the rack[45]. NFV orchestrator is responsible to estimate the warning time and communicate to the VNF managers and SDN controller.

[45] AWT need not be accurate, a rough estimate of around 40-50% of the total expected UPS discharge time is sufficient.

15.2.1.5 Snapshot and Restore Mechanism

Both the stateless and stateful VNFs are backed-up on stable nodes and only the stateful NFs need to be periodically updated with the state changes. VNF manager is responsible to periodically transfer and synchronize the the VNF states of the Transient VNFs with the standby VNF replicas. With multiplexed backup scheme, REARM optimizes the VNF state transfer by combining the state of all the VNFs that are replicated on the same remote node.

Multiplexed Backups: We make use of the N:1 backup model, such that more than one transient VNFs in GDC can be backed up on a single node in the brown energy powered data center. In general, our backup scheme across DC is N:M, where N>M. This configuration helps to trade performance with power for the duration the VNFs are expected to run on brown energy. With multiple VNFs backed on single node in SDC, ensures to save on brown energy, but at the cost of increased latency and lower throughput for short period of time. Once the power levels are back to normal operating conditions, NFVO triggers for the restoration of transient VNFs from the stable data center.

Addressing SFC: In order to account the service chains, we optimize by batching the updates for an entire chain of VNFs, so that the state of VNFs in a service chain is coherently synchronized and backed-up on stable services, instead of updating the state of each NF individually, which significantly reduces the communication overheads involved in state transfer.

15.2.1.6 Routing update

When both the transient VNF and standby nodes are in the same data center, the routing update is simple and incurs less overhead, as it only involves setting new set of forwarding rules to redirect flows towards the new instance. However, routing across data-centers is more complex and incurs delay in the order of seconds [173].

15.2.2 Design

Figure 15.2 illustrates the VNF state migration mechanism of REARM platform. We summarize below the key steps illustrating the working of REARM.

VNF Setup (1A, 1B): When the VNFs are provisioned, the NFVO sets up the active (Transient VNFs) on nodes powered by renewables and the associated back-up nodes. Depending on the DC power Infrastructure, these can be mapped either on different nodes in same data center or on GDC and SDC respectively. VNF Manager

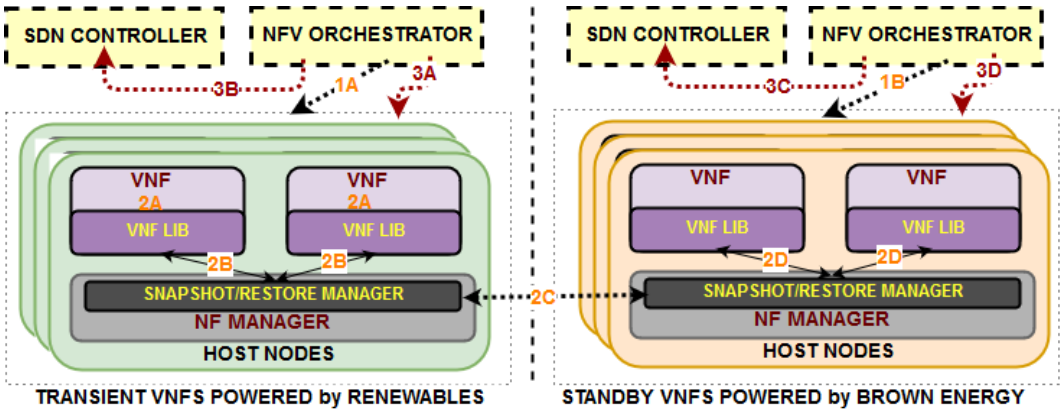


Figure 15.2: REARM's Operational steps for VNF migration.

on each of the host node is responsible to periodically extract the state updates of the transient VNFs and communicate to the VNF manager that hosts the standby VNF node. SDN controller is responsible to setup the forwarding rules for the flows that need to be processed by the VNF instances accordingly.

VNF State Tracking (2A, 2B): *libnf* periodically extracts (pull mode) the VNFs state change and sends the information to the NF Manager. Also, the VNF can explicitly trigger to communicate the state changes at the end of packet processing. This is useful to save critical state changes instantaneously, especially in the case of NFs that track per-flow state and update critical states during processing of intermittent packets like stateful firewalls and IDS.

VNF State Migration and updating (2C, 2D): NF Manager, upon receiving updates from the vnflib, transfers the state to the back-up nodes. Upon receiving the packets, the restore manager on the standby node identifies the corresponding VNF and notifies the vnflib to trigger for the state update.

Handling Downtime and restoration (3A-3D): Once the warning time is delivered by the UPS, the NFV Orchestrator (NFVO), communicates the warning time to all the VNF mangers and waits for the notification of state transfer completion. VNF managers perform best-effort approach to complete the state transfers within the specified warning time. Upon completion notification, or timeout, NFVO directs the SDN controller to setup and update the forwarding rules to redirect the traffic towards the VNFs in SDC. In case of LAN (within same data center), the SDN controller updates the forwarding rules in the switches to route the packets to the destination node. But, in the case of wide area networks (WAN), the Multi-Protocol Label Switching (MPLS) based VPNs are created as an abstraction for private net-

work address space shared across multiple data centers. NFVO in SDC also initiates the SDN controller and VNF managers on stable node to switch from standby to active mode. Once the power levels resume back to normal, the restoration from SDC to GDC is triggered and VNFs switch back their roles.

15.3 Implementation

We have implemented REARM using a DPDK-based NFV platform [46] for fast data plane processing, and leverage POX SDN controller to serve as REARM controller.

Table 15.1: VNFs used in our experiments

VNF	State update mode	State info type
Monitor (MON-0)	Stateless VNF	-
Monitor (MON-1)	per-flow	Global:128 B, Flow Table: 8 KB
Monitor (MON-2)	per-flow and per-packet	Global:128 B, Flow Table: 32 KB
Packet Logger (PLOG)	per-flow and per-packet	Global:128 B, Flow Table: 32 KB

We also implemented custom stateful VNF variants (Monitor and packet logger) that maintain per flow and per packet states, with aggregate state information size of 8KB and 32KB, supporting upto 1024 flows as shown in Table 15.1. Overall implementation[46] of REARM components NF Manager, *libnf*, excluding the custom VNFs is $\sim$ 1200 lines of code. All the VNFs link with the *libnf* that facilitates for VNF state import and export functionality.

Communication Mechanism: REARM provides a simple library called *libnf*, which abstracts the communication with the NF Manager from the VNF implementation. *libnf* is responsible to transfer and notify the VNF state updates with the NF Manager using the shared memory buffers. The *libnf* APIs allow the application code (VNFs) to export and import the VNF specific internal states are shown in Listing 15.3. At the time of initialization, VNF must register the callback function callback_fn() with *libnf* library. As the state characteristics are intrinsic and dis-

[46]We only account the implementation added to support VNF state migration and communication framework.

```
//Callback registration with vnflib
int vnflib_init(int nf_type, void (*callback_fn)(ip5tuple *flow_spec,
    state_type type, void *buf, size_t size));

//Pull mode API to Get NF's current state.
int pull_nf_state(ip5tuple *flow_spec, state_type type, void *buf, size_t
    size);

//Push mode API for NF's to update state.
int push_nf_state(ip5tuple *flow_spec, state_type type, void *buf, size_t
    size);

//Set the obtained VNF state to the VNF
int set_nf_state(ip5tuple *flow_spec, state_type type, void *buf, size_t
    size);
```

Figure 15.3: *libnf* APIs exported for facilitating VNF state transfers.

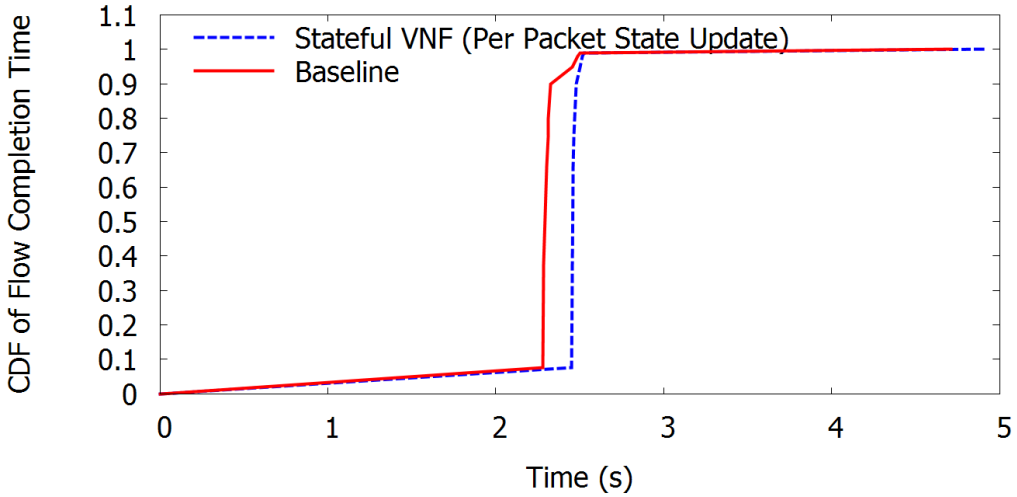
tinct for each VNF, the payload to export the state is treated as opaque pointer by *libnf*.

NF Manager implements the socket based communication protocol with the NF Manager on the remote nodes to transfer the VNF states. In order to accommodate variable sized state information of multiple VNFs to be transferred in a single packet, we package each VNF's state with tag-length-value format encapsulated in the UDP payload.

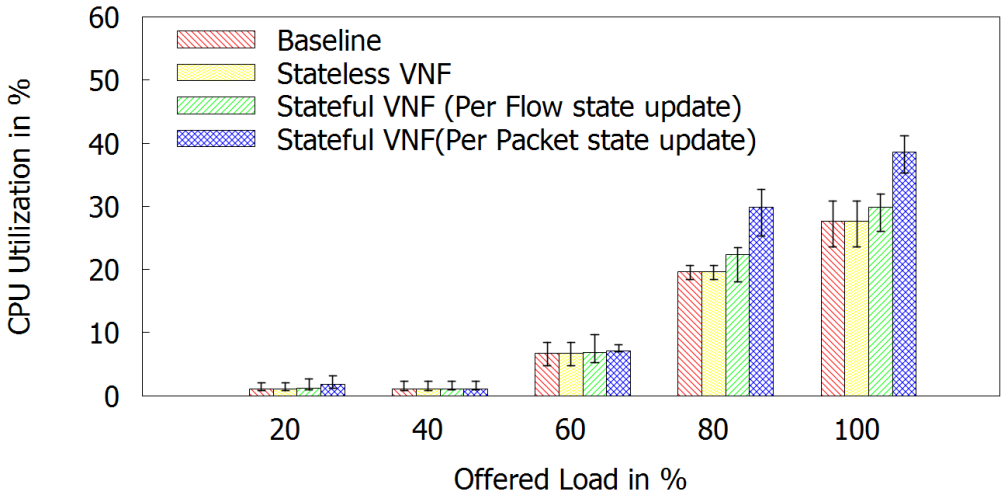
Adaptive Threshold and Push-Pull based State Transfer: *libnf* periodically (every 100us) pulls the VNF state and export to NF Manager. NF Manager buffers it until state of all NFs in the chain is obtained. But, when any VNF updates state with push based API, the state is immediately transferred to ensure that critical VNF states are synchronized. Threshold is dynamically computed by VNF Manager, which determines the total bytes of data that can be buffered and transferred for each service chain. Threshold computation is based on the number of service function chains served by VNF manager, measured round trip time to update VNF state on the replica, and previously issued AWT.

15.4 Evaluation

We performed preliminary evaluation on our university testbed. Our testbed has three Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz servers, with dual-port 10Gbps



(a) Overhead impact on Flow Completion Time of Web requests.



(b) CPU overhead for different VNF types.

Figure 15.4: Communication and Computation overhead analysis of REARM

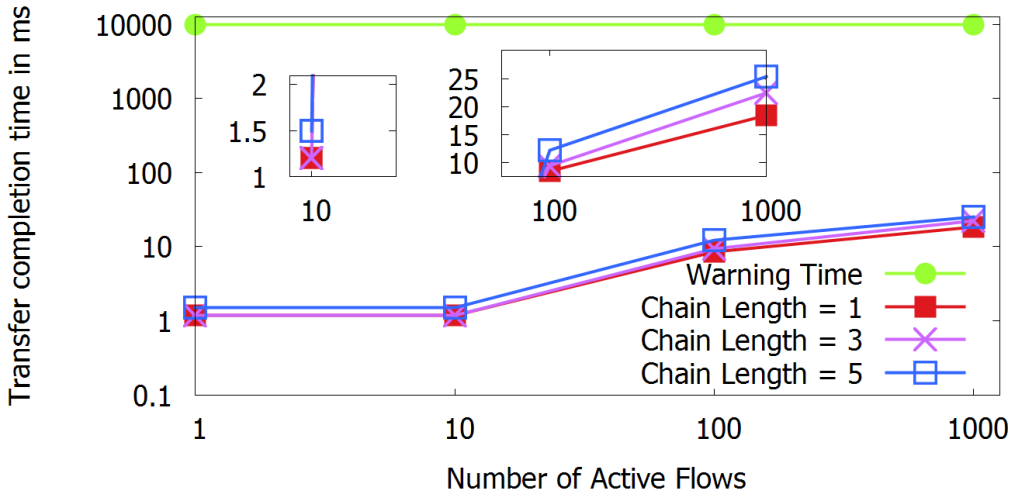


Figure 15.5: VNF migration time for different flows and chain lengths.

Table 15.2: Performance analysis using Apache bench, 10K web requests 32KB files with 500 concurrent requests

	Total Time	Mean Latency	Transfer rate
Baseline (Moniter)	47.119s	4.712ms	6842.08 KBps
Monitor (MON-2)	49.52s	4.952ms	6510.12 KBps

DPDK compatible NICs, running Ubuntu SMP Linux kernel 3.19.0-39-lowlatency. We designate one each for traffic generation, GDC node and SDC node respectively. We make use of Moongen [97] and Apache bench to generate line rate traffic and HTTP web traffic with varying numbers of flows.

Our primary focus of evaluation is i) to analyze the overhead of REARM in terms of processing cost and throughput, necessary to perform VNF state transfer and update the replica ii) to quantify the effectiveness of REARM for different service chain lengths and iii) demonstrate how quickly the VNFs can be replicated and restored on the remote nodes.

15.4.1 Overhead analysis

Latency and Throughput: First, we evaluate the latency and throughput overhead for running HTTP web requests using Apache bench. In this setup, we launch a total of 10K web requests, running 500 concurrent sessions, where each flow routed through a VNF, downloads 32KB file from the nginx web server. We compare VNF

that updates state for every processed packet with the baseline (without *libnf* state transfer). Table 15.2 shows the impact on mean latency (measured across all the concurrent sessions, has an increase of 240 μs), and the aggregate transfer rate (measured across all the web requests dropped by 4.8%) are very minimal.

Next, we run the same experiment on a chain of 3NFs *i.e.,* 1 VNFs with per-flow (MON-1) and 2 per-packet (MON-2 and PLOG) state update VNFs and capture the CDF of flow completion time as shown in Figure 15.4a. We can observe that the latency impact measured in terms of flow completion time is minimal and remains consistent without adding any additional tail latency *i.e.,* the relative flow completion time of 90th, 95th, 99th percentile of flows is not impacted.

Computation overhead: Figure 15.4b shows the CPU overhead incurred by VNF using the *libnf* for exporting the VNF state to the NF Manager. For this experiment, we use the variants of Monitor VNF (baseline) and compare with the stateless, per-flow and per-packet state update to estimate the computation cost overhead added in each of these cases for performing the VNF state transfers. We set packet size of 500 bytes and vary the load using the DPDK Pktgen tool, and record the average CPU utilization of the core pinned for executing the VNF using the mpstat tool.

We report the average CPU utilization reported over a period of 10 seconds for five different runs. We observe that CPU utilization for the stateless VNF is same as that of the Baseline VNF. And, compared to Baseline VNF, MON-2 (per-packet state update VNF) incurs highest CPU overhead, but even at 100% load (10Gbps), the average increase in CPU utilization is less than 5%, indicating the computation overhead of REARM is negligible.

15.4.2 NFV Resiliency and Warning Time Analysis

In Figure 15.5, we evaluate the VNF state transfer completion time for different service chain lenghts, after issuing the AWT, which is set to 10 seconds based on the solar and wind energy powered data center traces in [29]. Setup for chain length 1,3 are same as before, and we add two more VNFs (MON-1 and MON-2) for the chain length of 5. We launch long running flows and vary the number of flows served by VNF chain, and measure the time taken to completely transfer the VNF state.

We can observe that the state transfer time increases with the increasing number of flows, and is in the order of few (1-30) milliseconds. The increase is primarily due to state transfers performed for each flow. Second, with the increasing chain length for a given number of flows, we observe that the state transfer time remains

almost the same (only marginally increases) due to service chain based optimization. Also, we notice that 26 milliseconds is sufficient for 5NF chain serving 1000 flows to complete the VNF state transfer.

15.5 Conclusion

To summarize, we have characterized and analyzed the benefits and challenges in employing the Green Data Centers (GDCs) for VNFs. We have designed and implemented ***REARM***, especially to cater towards the special needs of VNFs migration, and introduced the concept of Transient VNFs, which can be sufficiently restored to stable powered nodes within the specified advance warning time.

With prototype based evaluation on our SDN/NFV testbed, we have demonstrated the potential benefits of incorporating REARM towards achieving high availability and resiliency with VNFs, with the known warning times for the Green Data Centers. REARM exhibits extremely low computation and communication overheads (less than 5%) and able to efficiently migrate VNF states for service chains serving thousands of flows in less than 30 milliseconds.

Chapter 16

Future Prospects

In this chapter we present the possible areas of extension of our work. We consider and present the future prospects of the proposed NF and NSC failure resiliency mechanisms of REINFORCE and DR schemes of REARM, in the view of other NFV platforms and technology advancements.

Contents

16.1 Recap on resiliency framework

Table 16.1 compares the supported resiliency features, performance and overhead characteristics for different state-of-the-art works.

16.2 Current Limitations and Prospects of Extensions

We acknowledge that the network routing and communication latency for migrating the VNFIs across *Wide Area Network* (WAN) would be more demanding and challenging to address. Hence, we plan to extend our evaluation to extensively study the associated trade-offs and conduct cross-site and large scale data center topology based evaluation with real traffic traces.

Table 16.1: Comparison of the related state-of-the-art solutions with REINFORCE for NF and NF Chain Resiliency.

Work/Feature	Pico Replication	FTMB	Stream NF	Stateless NF	REINFORCE
High Level Architecture & Approach					
FT scheme	Application level Check-Pointing	Replay + VM Check-Pointing	Externalize state + Replay	Externalize state <No replay>	Application level checkpointing + Replay
Redundancy scheme	Active-Standby (1:N)	Active-Standby (1:1)	Active-standby M:1	Active-standby M:1	Active-Standby(1:N)
Recovery scheme	Only reroute flows by SDN controller; No Replay. state is check pointed.	Replay mode: Replay packets from Input logger + PAL from Output Logger	Replay mode: Replay packets from Root node + play NF in replay mode	Re-route flows to replica instance	Hierarchical Failover Local reroute by NFVM Remote reroute and Replay by VNFM
Instrument MB code	Minimal	High†	High	Minimal	Minimal
Failure Detection	SDN Port_down & connection down	Not Addressed	Not Addressed	Not Addressed	Local Monitoring and per link BFD
Failure Types	Soft (NF) Failure: ✓ Link Failures: ✓ Node Failures: ✓	Soft (NF) Failure: ✓ Link Failures: ✕ Node Failures: ✕	Soft (NF) Failure: ✓ Link Failures: ✓ Node Failures: ✓	Soft (NF) Failure: ✕ Link Failures: ✕ Node Failures: ✕	Soft (NF) Failure: ✓ Link Failures: ✓ Node Failures: ✓
Checkpoint Freq/s	1000	5-50	–	–	10-100
Service chain	NO*	NO*	YES	NO	YES
Failure Free Overheads					
Latency Overhead (ppktmedian,99%ile)	15-20% (2-3ms) 5sec?	30us 35ms	2-3us 13ms	65-300us 1ms?	10-200us
Throughput impact (% drop)	3-5% 50% due to pkt copy	5-30%	Not Reported	∼15-20%‡	∼2-15%
Measured Packet/Data rate	10-200Mbps 10-200KPPS	1-10Gbps* 1.4-4 Mpps	Not Reported	1-8Gbps ∼4.7Mpps	2-10Gbps ∼12Mpps
Service Unavailability	RTT to SDN controller	Not specified	Not specified	Not specified	5*Link Delay
Restoration Time	Order of milli sec.	40-300 ms	Not specified	Not Applicable	5-10 ms

†: Beyond the need to annotate and wrap shared variable access pattern, special synchronization logic is required for replay.
‡: The measured throughput is for both input logger, output logger, active and standby middleboxes running on a single node. with cross node repliction, it is expected to be much lower.
*: By design they can support service chain provided the replication is performed for each of the middlebox in the chain.

16.3 Applicability of REINFORCE in other NFV Platforms

The REINFORCE approach of distinguishing the minimal essential information that need to be migrated *i.e.,* packet-processing-progress and lazy sync of actual NF state of all NFs in the chain is a generic mechanism. It should be easy to impart these mechanisms in any platform.

16.3.1 ClickOS

In ClickOS framework, NF chain is realized by distinct click elements. It is possible to implement and insert click-elements at the start and end of distinct phase to ensure correct time stamping and to capture the packet-progressing progress for each NF. To achieve this, the hypervisor needs to be extended to monitor and track the chain-wide processing progress and perform NF state synchronization.

16.3.2 NetBricks

In NetBricks, the NF chain is realized by composing multiple NFs in to a single executable where the action of each NF is merely a statically defined function call. In this run-to-completion model, it is hard to separate the execution boundary of each NF and to trap the exact progress of each NF. Hence, ensuring the correctness at each NF boundary is not possible. However, when entire NF chain is treated is a single composite NF, then it makes it easier to apply the REINFORCE mechanism for both local and remote replications.

Chapter 17

Conclusion

The more we learn, the more we understand the vastness of our ignorance. We can only, but humble and cherish our drop of knowledge in the fathomless ocean of ignorance.

— Author

This thesis presents the NFV resource management framework in the realm of ETSI NFV-MANO reference architecture. The key components of this work have addressed the system and network wide performance, scale and reliability challenges associated with the deployment of the NF chains.

Contents

17.1 Dissertation Summary

First, we studied the performance and scale challenges associated with deploying the softwarized NFs on COTS hardware and presented the novel rate-cost proportional scheduling for user-space NF platforms like DPDK. We also designed the chain-aware backpressure mechanism that ensures to avoid the wasted work across the NF chain and facilitates judicious resource allocation to the NFs for both the chains contained within a same-core or spanning multiple cores. The functionalities

facilitate towards realizing the VNFM and EMS components of the NFV-MANO architecture.

Second, we studied the network-wide NFV platform characteristics and presented the novel semi-distributed resource management framework to address the traffic characteristics and perform NF placement, network congestion, and load balancing the flows among the active NFs. We model the semi-distributed framework, where SDN controller and VNFMs take co-ordinated decisions to achieve optimal NF placement and balance the load in the network. The functionalities facilitate towards realizing the NFVO, and SDN Controller components of the NFV-MANO architecture.

Finally, we developed the resiliency framework for NFV platform. We presented the novel NF state replication strategy and distinct mechanisms to provide timely detection of NFs, hardware node (VNFM), and network link failures. Further, we implemented distinct failover mechanisms with strict correctness guarantees. We also presented the mechanism to provide service continuity by overcoming the stability concerns with GDCs. The resiliency functionalities facilitate towards realizing the NFVI, VNFM, NFVO, EMS and SDN Controller components of the NFV-MANO architecture.

All the corresponding source code and platform implementations are open-sourced and made available online.
Part I (NFVnice):
`https://github.com/sameergk/openNetVM-dev/tree/clean_nfvnice`[47]

Part II (DRENCH):
`https://bitbucket.org/sameergk_ugoe/drench`

Part III (Resiliency Framework):
`https://github.com/sameergk/openNetVM-dev/tree/nfv_res`[47]

17.2 Dissertation Impact

The contents of this dissertation have been published in the following peer-reviewed international conference proceedings:

Preliminary versions of Chapters in part I (*i.e.,* Addressing system level challenges with the deployment of Network Service Chains) appear in the paper:

[47] Access to OpenNetVM-dev branch needs to be requested; The code shall be forked to a public branch of OpenNetVM soon.

(i) ***NFVnice*** [30] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 71–84, New York, NY, USA, 2017. ACM

Preliminary versions of Chapters in part II (*i.e.,* Addressing Network wide challenges the deployment of Network Service Chains) appear in the papers:

(ii) ***DRENCH*** [31] Argyrios G. Tasiopoulos, Sameer G. Kulkarni, Mayutan Arumaithurai, Ioannis Psaras, K. K. Ramakrishnan, Xiaorning Fu, and George Pavlou. DRENCH: A semi-distributed resource management framework for NFV based service function chaining. In *2017 IFIP Networking Conference (IFIP Networking)*. IEEE, June 2017

(iii) ***NeoNSH*** [32] Sameer Kulkarni, Mayutan Arumaithurai, K. K. Ramakrishnan, and Xiaoming Fu. Neo-NSH: Towards scalable and efficient dynamic service function chaining of elastic network functions. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE, March 2017

(iv) ***NSN*** [37] S. G. Kulkarni, M. Arumaithurai, A. Tasiopoulos, Y. Psaras, K. K. Ramakrishnan, Xiaoming Fu, and G. Pavlou. Name enhanced sdn framework for service function chaining of elastic network functions. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 45–46, San Francisco, USA, April 2016

Preliminary versions of Chapters in part III (*i.e.,* Addressing Failure Resiliency for Network Service Chains) appear in the paper and technical report:

(v) ***REARM*** [34] Sameer G Kulkarni, Mayutan Arumaithurai, K.K. Ramakrishnan, and Xiaoming Fu. REARM: Renewable energy based resilient deployment of virtual network functions. In *2017 European Conference on Networks and Communications (EuCNC)*. IEEE, June 2017

(vi) ***REINFORCE*** [35] S. G. Kulkarni, G. Liu, K. K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, (2018). REINFORCE: Achieving Efficient Failure resiliency for Network Function Virtualization based Services. Addressing nfv resiliency. `https://github.com/sameergk/Addressng-NFV-Resiliency`, 2018. [ONLINE][48].

[48] *This work is submitted to CoNext'18 and is currently under review.*

Building on the work listed above, the author has further identified and supervised the following topics for Master theses:

(a) "Evaluation of state of the art works on SDN/NFV Placement and Load Balancing", Hari Raghavendrarao Bhandari, University of Göttingen, Masters Thesis 15-May-2017.

(b) "NSH Routing: Implementation of Network Service Headers to realize the service chain by steering traffic across the VNFs" , Guido Martinez, University of Göttingen, Masters Thesis 15-Jan-2018.

17.3 Future Prospects

This dissertation has tried to address a few of the key *Network Service Chaining* (NSC) related problems, namely ***P1***:*Performance Optimization*, ***P2***:*Management and Orchestration* and ***P3***:*Availability and Reliability*, towards our goal of making the NFV deployment a reality by addressing specifically the performance, scale and reliability concerns outlined in section §1.1.1.

17.3.1 Extensions to the current work

The other outliend high level research problems, especially the problems relating to ***P4***:*Security, Policy and Trust Management* and ***P5***: *Interoperability and Portability* need specific attention.

The recent surveys [14, 174, 175] on why the NFV adoption has not been able to meet the hype predicted in early 2014/15, have identified the key technical and non-technical hindrances/potential barriers as follows[49]:

- Technical Problems:
  - Existing platform and VNF Interoperability
  - Security concerns with VNFIs
- Non-Technical Problems:
  - Maturity/stability of NFV platform
  - Lack of expertise and training on NFV

As the open source NFV projects are quite new, the non-technical concerns are

[49]These concerns strongly overlap with our initial outlined problems and the challenges that we deemed important for successful deployment of NFV.

justified and would definitely be resolved with the time and maturity of new NFV platforms. However, the key technical problems, which overlap with our earlier outlined Problems ***P4*** and ***P5*** need to be addressed.

ETSI's NFV-ISG and IETF NFV and SFC working groups have significantly contributed towards outlining the key specifications and API requirements necessary to address the ***P5***: *Interoperability and Portability*.

Our next prospective is to address the open problem ***P4***: *Security, Policy and Trust Management*, and also to retrofit and enhance on the achieved goals in the current state-of-the-art market strategies.

17.3.2 Broader Future Directions

The key networking technologies where SDN and NFV can play an active role towards the development and to potentially shape the future networking landscape include:

- 5th generation wireless systems (5G),
- Internet of Things (IoT),
- Edge Cloud and Fog computing,
- Microservices

Further research in identifying and addressing the compelling problems and associated challenges that can leverage SDN and NFV, and aid towards enabling these future technologies to be deployable, efficient, reliable and robust is necessary.

17.3.2.1 Role of SDN and NFV in 5G, IoT, and Edge Computing

Fifth-generation technology (5G), has already gained lots of research attention. The first commercial solution is expected to be demonstrated by 2020, while the standards framework for 5G communication networks is already in place and expected to be finalized by early 2019. Also, by 2020, it is predicted that there will be over 50 billion connected (IoT) devices worldwide. The main objectives of 5G networks is to enable ultra low latency services (less than 1 millisecond), and achieve 1000x improvement in network capacity (10-100Gbps bandwidth).

For the success of 5G, it is necessary to dynamically and optimally provision and manage network resources accounting both maximal cost-efficiency and diverse 5G service demands. Hence, 5G defines network slicing as an End2End logically self-contained pool of network resources, spanning across the core and access networks.

Ensuring such a network slice exhibits both network and system level challenges. It demands a lot of consideration on several aspects including resource provisioning, protocols and standardized interfaces to facilitate cross network communication, service aware slice management framework, slice isolation, partitioning and grouping mechanisms *etc.*.

There is already active research going on in this context and lot can be leveraged from Cloud-RAN, Deutsch Telekom, and Telefónica. However there are several performance, management and orchestration issues that still demand greater attention. Hence, a logic extension to my current research would be to address and present a holistic framework for network slicing by accounting the Edge, access and core networks through SDN and NFV solutions.

17.3.2.2 Towards reliable and scalable services through Microservices

On the other hand, the rise of microservices architecture in networking and cloud based applications, along with increased enthusiasm on service automation through the employment of machine learning and deep learning technologies that profess reliance on big data analytics continue to change the data center traffic patterns within and across data centers.

Though microservice architecture is best-fit for application scaling and building reliable and robust services, they come at a cost of increased communication overhead and latencies. Also, considering the strict 5G latency requirements, massive scale of IoT, and diverse communication use cases, it is necessary to perspicuously look into how far and to what extent micro-services can be employed and what strategies can be applied to leverage and sustain the benefits of micro-services without compromising on the service guarantees. Hence a holistic software framework or a decision tool, that can account both the architectural constructs and run-time system characteristics including CPU, memory, I/O and communication requirements of the software in building micro-services is necessary.

Such a work would enable to rightly decompose a monolithic service into microservices and enable to leverage the key benefits of miro-service architecture to build and deploy highly scalable, flexible and resilient network services without jeopardizing the overall performance and service level constraints.

Part IV

Appendix

Table of Contents

The appendix complements the dissertation with more detailed descriptions of algorithms and examples that we felt were too detailed, complicated, or formal for the main text. Nevertheless, the contents are important results of the dissertation and form a considerable part of the overall contribution.

Chapter A

Concepts and Definition of Related Terms

A.1 Concepts and Definitions

Address Space Layour Randomization (ASLR) is a mechanism to randomize the location of memory where the system executables are loaded. It is a means of memory-protection for OSes to guard against the buffer-overflow attacks.
Note: *With DPDK, it is recommended to disable ASLR (but not necessary), as the position of the hugepage (and other) memory in the DPDK primary process or secondary process virtual address space can change across different runs resulting in conflicting views if the ASLR is enabled.*

Gi-LAN The segment of network in telecommunication networks where the service providers deploy *Transmission Control Protocol* (TCP)/*Internet Protocol* (IP) functions between the packet gateway and the Internet. This section of network is where CSPs typically innovate, differentiate, and monetize services using unique capabilities through a combination of homegrown solutions and those provided by a wide variety of suppliers.

LLMV *Low-Level Virtual Machine* (LLVM) is a library for programmatically creating machine-native code. A developer uses the API to generate instructions in a format called an intermediate representation (IR). LLVM can then compile the IR into a standalone binary, or perform a *Just-in-time* (JIT) compilation on the code to run in the context of another program, such as an interpreter for the language.

Middlebox refers to a network appliance specifically to any intermediary device (placed in the path between a source host and destination host) performing functions other than the normal packet forwarding. It can either transform, inspect, filter,

or otherwise manipulate the traffic. Generally these are purpose build hardwares *e.g., Load Balancer*, *Network Address Translation*, Firewall, *Wide Area Network* Optimizers *etc.*

Network Function NF refers to a functional block within a network infrastructure that has well-defined external interfaces and well-defined functional behaviour. NF can be either a physical compute node *i.e.,* PNF or a virtual node *i.e.,* VNF.

Network Function Orchestrator NFVO refers to the functional block that manages the network service lifecycle and coordinates the management of VNF lifecycle with the *Virtualized Network Function Manager* and *Network Functions Virtualization Infrastructure.* resources (supported by the VIM) to ensure an optimized allocation of the necessary resources and connectivity [176].

Network Processors A network processor NPU is a programmable software device (integrated circuit) used as a network architecture component inside a network application domain. It is analogous to CPU in a computing devices.

Network Service refers to a composition of network functions *i.e.,* one of more VNFs and/or PNFs. The specifics of the network functions is governed by the functional and behavioral specification of the service. The network service is an end-to-end orchestration that map to the operator or network policies, and is characterized by at least performance, dependability, and security specifications.

Network Utility Maximization: is a mathematical linear programming optimization formulation for resource allocation problem especially for the purpose of flow control (*i.e.,* the network bandwidth allocation to flows). It can be formulated as an optimization problem whose objective is to maximize an aggregate utility function of all nodes/sources/users of the network while subject to some constraints regarding the limited capacity of each network's link [177].

Service Function Chaining refers to the definition and instantiation of an ordered set of service functions (*i.e.,* PNFs and/or VNFs) and subsequent steering of traffic through them to realize one or more NSs [21]. Note: It is also known as ***Network Service Chaining*** and ***VNF Forwarding Graph***.

Shadow Price refers to the value of the Lagrange multiplier at the optimal solution. In general, it corresponds to the estimated value that determines the marginal utility or cost of a specific constraint variable.

Chapter B

NFVnice Algorithms and Workflow

B.1 CGroup Setup

Enabling Cgroup in Linux: In file: `/boot/config-'kernel.version'`

```
CONFIG_CGROUPS=y
CONFIG_CGROUP_SCHED=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CPUSETS=y
CONFIG_BLK_CGROUP=y
CONFIG_PROC_PID_CPUSET=y
CONFIG_CGROUP_FREEZER=y
```

Listing B.1: Enabling CGroup in Linux Kernel

Check and Mount the cgroup filesystem

```
#if [! -d "/sys/fs/cgroup/" ]; then
    mount -t cgroup none /sys/fs/cgroup
fi
```

Listing B.2: Enabling CGroup in Linux Kernel

B.2 Tuning CFS

The following CFS parameters need to be tuned to ensure low latency context switches[50]

```
#echo 100000 > /proc/sys/kernel/sched_min_granularity_ns
#echo 1000000 > /proc/sys/kernel/sched_latency_ns
#echo 25000 > /proc/sys/kernel/sched_wakeup_granularity_ns
```

Listing B.3: Configuration of CFS Parameters

[50] sched_latency_ns is the period of 1 round; sched_min_granularity_ns determines the minimum preemption granularity; sched_wakeup_granularity_ns to tune preemption lag.

B.3 Algorithms and Pseudocode

1)Data structure extensions for service-chain specific back-pressure variables:

```
struct onvm_service_chain {
    struct onvm_service_chain_entry sc[ONVM_MAX_CHAIN_LENGTH];
    uint8_t chain_length;
    int ref_cnt;
#ifdef ENABLE_NF_BACKPRESSURE
// Flag indicating the Chain has one or more Downstream NF overflowing.
    uint8_t downstream_nf_overflow;
// Bit index of each NF in the chain that is overflowing
    uint8_t highest_downstream_nf_index_id;
//Flag is set when all nf_instances are populated in the below array
    uint8_t nf_instances_mapped;
// NF instances serving this chain
    uint8_t nf_instance_id[ONVM_MAX_CHAIN_LENGTH];
#endif //ENABLE_NF_BACKPRESSURE
};
```

Listing B.4: Extensions to account Backpressure per Service chain.

```
struct client {
    struct rte_ring *rx_q;
    struct rte_ring *tx_q;
    struct onvm_nf_info *info;
    uint16_t instance_id;
#ifdef ENABLE_NF_BACKPRESSURE
    uint8_t throttle_this_upstream_nf; // Flag indicating if this NF needs to be Throttled
    uint64_t throttle_count; // Counter tracking the num of throttles.
#endif //ENABLE_NF_BACKPRESSURE
};
```

Listing B.5: Backpressure state extension to NF specific structure

```
// HIGH WATERMARK LEVELS FOR NFs Rx Ring Buffers
#define CLIENT_QUEUE_RING_THRESHOLD (80)
#define CLIENT_QUEUE_RING_WATER_MARK_SIZE ((uint32_t)
((CLIENT_QUEUE_RINGSIZE*CLIENT_QUEUE_RING_THRESHOLD)/100))
// LOW WATERMARK THRESHOLD FOR NFs Rx Ring Buffers
#define CLIENT_QUEUE_RING_THRESHOLD_GAP (20)
#define CLIENT_QUEUE_RING_LOW_THRESHOLD ((
    CLIENT_QUEUE_RING_THRESHOLD >
CLIENT_QUEUE_RING_THRESHOLD_GAP) ? (
    CLIENT_QUEUE_RING_THRESHOLDCLIENT_
QUEUE_RING_THRESHOLD_GAP):(CLIENT_QUEUE_RING_THRESHOLD))
#define CLIENT_QUEUE_RING_LOW_WATER_MARK_SIZE ((uint32_t)
((CLIENT_QUEUE_RINGSIZE*CLIENT_QUEUE_RING_LOW_THRESHOLD)/100))
```

Listing B.6: Queue size and Backpressure threshold definitions

2) Logic to Detect and apply backpressure for any NF in the chain.

```
IF (−EDQUOT OR −ENOBUFS == rte_ring_enqueue_bulk()) THEN %AND (
     NF_SVC_ID is not the first in chain) THEN
    FOR EACH packet in the Enque_Packets
        GET associated service_chain
        SET service_chain−>downstream_nf_overflow = TRUE
        SET BIT service_chain−>highest_downstream_nf_index_id, meta−>index
        FOR Index=1; index < meta−>index; in service_chain
            client [schain−>nf_instance_id[index]]−> throttle_this_upstream_nf = TRUE
        END FOR
    END FOR
END IF
```

Listing B.7: Rx/Tx Threads detect and mark NF in a chain for backpressure

3) For NF Schedule Throttling: i) In the NF_LIB always check for the block_flag set state and voluntarily block on the semaphore, after processing every batch of packets.

```
IF (rte_atomic16_read(BLOCK_NF_FLAG) ==1) THEN
    CALL onvm_nf_yeild()
END IF
```

Listing B.8: *libnf* logic to de-schedule the upstream NFs on NF chain backpressure

ii) NF_Manager wakeup-thread determines whether the NF needs to be blocked.

```
IF (nf_instance−>throttle_this_upstream_nf is TRUE) THEN
    SET BLOCK_NF_FLAG=TRUE
ELSE
    CALL WAKEUP_NF_INSTANCE()
END IF
```

Listing B.9: wake-up thread Logic to block the NFs

4) For Packet Throttling: i) In the onvm_pkt_enqueue_nf() Rx/Tx threads check for the downstream_nf_overflow flag status and selectively drop the packets for each service chain.

```
IF (s_chain−>downstream_nf_overflow is TRUE) AND (is_upstream_NF(nf_instance))
     THEN
    CALL onvm_drop_pkt()
    nf_instance−>stats.rx_drop++
    nf_instance−>throttle_count++;
ENDIF
```

Listing B.10: Selective packet drops for back-pressured chains by Rx/Tx threads.

5) Stop back-pressure mode: Detect and Unblock NFs
On dequeues buffer, Tx thread checks for the low_watermark level.

```
IF (RING_LOW_WATERMARK_SIZE >= rte_ring_count()) THEN
    FOR EACH packet in the Dequeued_Packets
        GET associated s_chain
        IF s_chain->downstream_nf_overflow = TRUE THEN
            TEST & CLEAR BIT s_chain->highest_downstream_nf_index_id,meta->
    index
            FOR Index=1; index < meta->index; in service_chain
                nf_instance[schain->nf_instance_id[index]]-> throttle_this_upstream_nf
    = TRUE
            END FOR
        END IF
        IF s_chain->highest_downstream_nf_index_id = 0 THEN
            s_chain->downstream_nf_overflow = FALSE
        END IF
    END FOR
END IF
```

Listing B.11: Tx thread Clearing the Backpressure

B.4 Work Flow Diagrams

B.4.1 Workflow for Asynchronous I/O (read) operation

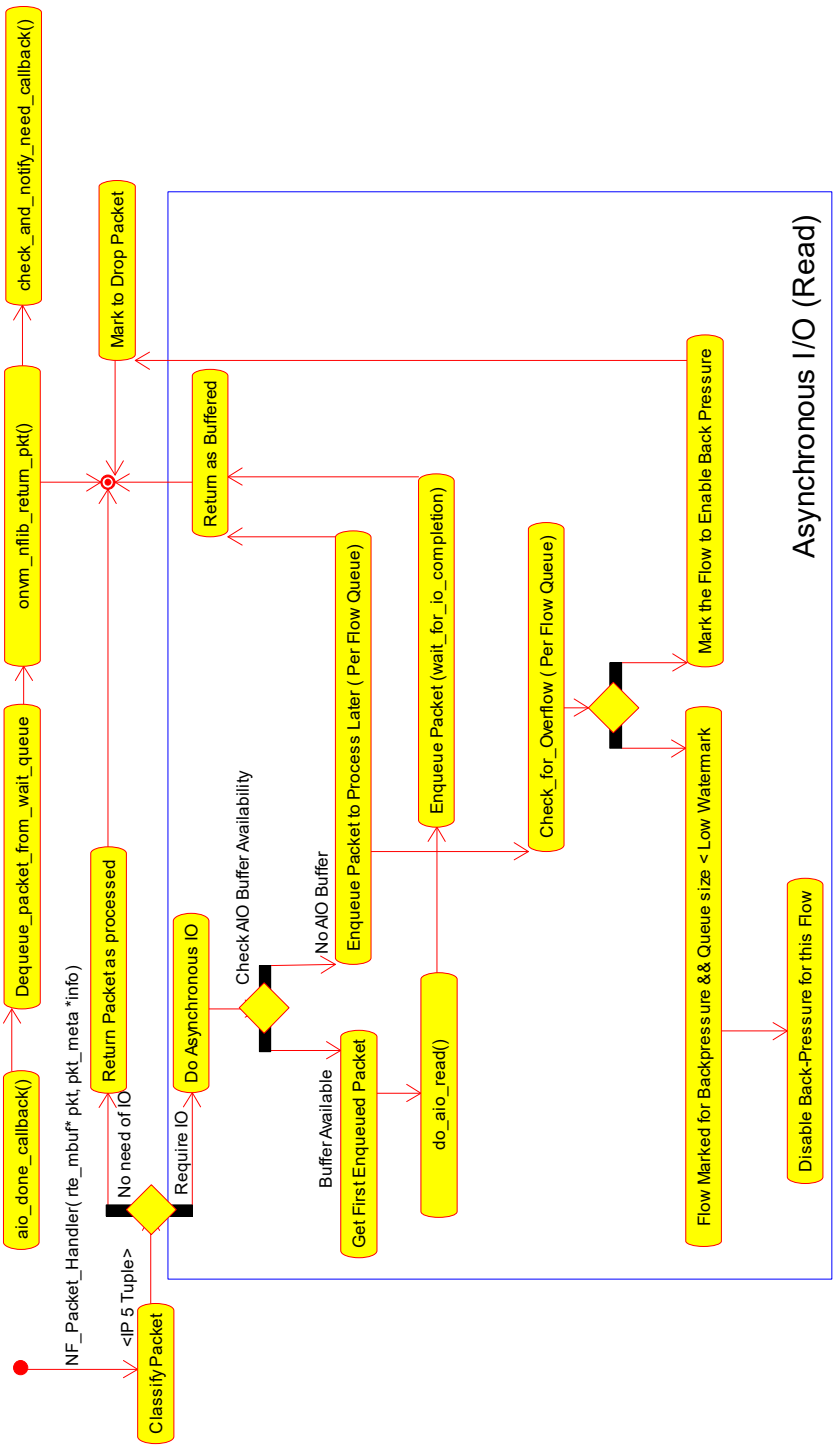


Figure B.1: Work flow for performing Asynchronous I/O read operation for selected incoming packets with optional support to classify and enable per flow queuing.

Chapter C

REINFORCE Proof of Correctness, Algorithms, and Workflow

C.1 Proof of Correctness

C.1.1 NF Packet Processing Model and Notions

We consider Network Functions (NFs) to be represented as finite state machines, that process stream of incoming packets p_{ij} (*i.e.* i^{th} *packet of* j^{th} *flow*) and as a result of processing packets they update or transition their internal state and output one or more packets p'_{ij} as shown in Figure C.1. For the NF Chain, the packets are sequentially processed by distinct NFs in the chain resulting in distinct state update at each of NFs and output packets.

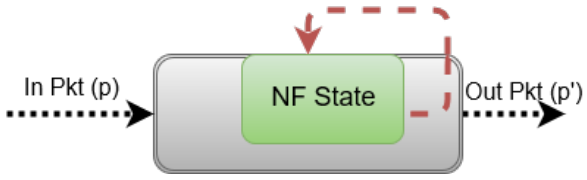


Figure C.1: NF Packet Processing and State Machine Abstraction

In REINFROCE, we optimize the performance of NF processing chain by accounting External Synchrony that allows to conditionally operate NF chains in asynchronous mode (without the need to commit the NF state across replica), with minimal amount of state transfer which tracks the packet processing progress of different flows.

We begin by providing some of the basic Definitions of packet processing, external

$\mathbb{B}$	Set of Buffered packets at the predecessor node $= (p_0, p_1, p_2, \ldots, p_x)$
$\mathbb{P}_i$	Set of processed packets at the Primary NF (i) $= (p'_0, p'_1, p'_2, \ldots, p'_n) \ni \mathbb{P}_i \subseteq \mathbb{B}$
$\mathbb{S}_i$	Set of abstract states of NF (i) $= (S_0, S_1, S_2, \ldots, S_x)$
$\mathscr{T}_j$	Set of Timestamp of the last packet of flows (j) committed to secondary $= (t_{ij}) \ni \mathscr{t}'_i forflow(j)$
$\mathbb{R}$	Set of packets released by Primary node presenting the external view. $= (p'_0, p'_1, p'_2, \ldots, p'_m) \ni \mathbb{R}_i \subseteq \mathbb{P}_i \subseteq \mathbb{B}$
$\mathscr{P}.\mathscr{V}_i$	State of Primary NF (i) representing the primary view $= \mathbb{P}_i * S_i \mapsto S'_i$
$\mathscr{P}.\mathscr{V}$	Set of States of all Primary NFs representing the primary view of node $= (\mathscr{P}.\mathscr{V}_0, \mathscr{P}.\mathscr{V}_1, \mathscr{P}.\mathscr{V}_2, \ldots, \mathscr{P}.\mathscr{V}_i)$
$\mathscr{E}.\mathscr{V}_i$	External View of the state at Primary NF(i). $= \mathbb{R} * S_i \mapsto S'_i$
$\mathscr{E}.\mathscr{V}$	Set of States of all Primary NFs representing the External view of node $= (\mathscr{E}.\mathscr{V}_0, \mathscr{E}.\mathscr{V}_1, \mathscr{E}.\mathscr{V}_2, \ldots, \mathscr{E}.\mathscr{V}_i)$
$\mathscr{S}.\mathscr{V}_i$	State at Secondary NF(i) committed by Primary. $= \mathscr{S}.\mathscr{V}i \subseteq \mathscr{P}.\mathscr{V}_i$
$\mathscr{S}.\mathscr{V}$	Set of States of all Secondary NFs representing the secondary view of replica node $= (\mathscr{S}.\mathscr{V}_0, \mathscr{S}.\mathscr{V}_1, \mathscr{S}.\mathscr{V}_2, \ldots, \mathscr{S}.\mathscr{V}_i)$

Table C.1: Notations used for Correction Analysis.

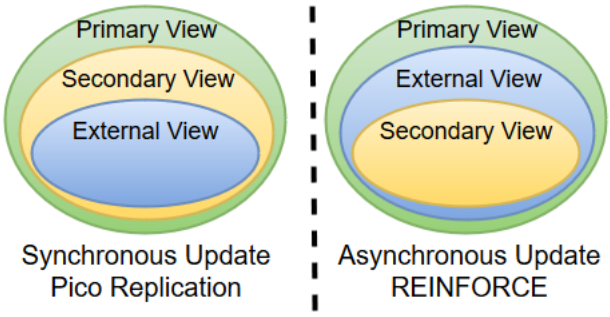


Figure C.2: Relationship of NF States across Primary, Secondary (Replica) and External observer (Client view)
With Synchronous update (*e.g.,* Pico Replication), the External view is a subset of Replica
With Asynchronous update (*e.g.,* Deterministic updates in REINFROCE), the Replica is subset of External view.

synchrony and the assumptions that hold good for NF and NF chain wide packet processing in REINFORCE.

C.1.2 Definitions and Assumptions

We consider the following definitions and assumptions from previous literature.

Definition C.1 (Deterministic Processing) For given NF State, processing a packet p_i always results in same deterministic state transition $P * S_i \mapsto S_j$ and deterministic output p'_i.

Definition C.2 (Non-Deterministic Processing) For given NF State, processing a packet p_i may result in different state transitions $P * S_i \mapsto \{S_i\}$ and yield different output $\{p'_i\} \subseteq (p'_0, p'_1, p'_2, \ldots, p'_i)$.

Definition C.3 (External Synchrony) State Synchrony is defined by externally observable behavior and not the actual internal system (NF or Chain of NFs) state, and on failover, the externally observable state remains consistent without regards to the actual consistency of internal NF states across replicas.

Theorem C.1 (Correctness of Operation) For remote replication, REINFORCE preserves external synchrony and ensures correctness of operation for Network Function chains.

Additionally, we consider the following two assumptions based on the packet processing behavior of NFs:

Assumption (Duplicate Packet Processing) *Duplicate packet processing by any Network Function (NF) results in same output as that of original packet without impacting the internal NF state.*

Assumption (Correctness Criteria) *External Synchrony is necessary and sufficient to preserve the correctness of the operation of Network Functions.*

Note that the Assumption C.2 is already proven by Nightangale *et.al.* [162, 178], and the basis of our Assumption C.1 is illustrated in §14.2.

C.1.3 Proof

We prove Theorem C.1 by the methods of "Proof by case" and "proof by contradiction". We know that the packet processing by an NF can result in either i) non-deterministic or ii) deterministic state updates in an NF, and accordingly in REINFORCE replication is based on the following two propositions.

Proposition (Packet Processing Progress) *To preserve external synchrony, in the case of only deterministic state updates, it is sufficient to track and update the packet processing progress information across NF chain.*

This implies that REINFORCE only updates the packet processing progress information $\mathscr{T}_j$ before releasing the packets that update the external view and then lazily (periodically) updates the NF state to the replica node much later than release of the packets. Hence as shown in the right side of the Figure C.2,

$$\mathscr{S}.\mathscr{V}i \subseteq \mathscr{E}.\mathscr{V}_i \ni \mathscr{E}.\mathscr{V}i \subseteq \mathscr{P}.\mathscr{V}_i and \mathbb{P}_i \subseteq \mathbb{B} \tag{C.1.1}$$

Proposition (External Synchrony with Non-Deterministic processing) *In order to preserve, external synchrony in the event of non-deterministic packet processing, it is necessary to synchronize and commit the NF state at replica $\mathscr{P}.\mathscr{V}_i$ before releasing the packet p'_i.*

This implies that REINFORCE updates the NF state to the replica node and only then releases the packets. Hence as shown in the left side of the Figure C.2,

$$\mathscr{E}.\mathscr{V}i = \mathscr{S}.\mathscr{V}_i \ni \mathscr{S}.\mathscr{V}i \subseteq \mathscr{P}.\mathscr{V}_i \tag{C.1.2}$$

Case 1: Let us consider the case when the primary node fails given that packet processing (initial or after last packet processing progress commit) update $\mathscr{T}_j$ resulted in only deterministic state update in any of the NF in the chain. In such case, By the Definition C.1, reprocessing of any $t_{ij} \in \mathscr{T}$ packets in $\mathbb{B}$, only updates the NF state at the secondary NF, but these packets are subsequently dropped by the Replica node and hence do not modify the $\mathscr{E}.\mathscr{V}$. However, the secondary NF(i) processes the packets, which implies: $\mathscr{S}.\mathscr{V}_i = \mathbb{P}_i * S_i \mapsto S_i'$ resulting in $\mathscr{S}.\mathscr{V}_i = \mathscr{P}.\mathscr{V}_i$
If we consider the contradiction that $\mathscr{S}.\mathscr{V}_i \neq \mathscr{P}.\mathscr{V}_i$, then it violates the Definition C.1, which cannot be true.
And, for the remaining $[\mathbb{B} - \{\mathscr{T}_i j\}]$ packets, the $\mathscr{P}.\mathscr{V} = \varnothing$ and as per Proposition C.1 $\mathscr{E}.\mathscr{V} \subseteq \mathscr{P}.\mathscr{V} = \varnothing$. Hence, in either case the external synchrony $\mathscr{E}.\mathscr{V}$ is preserved.

Case 2: Lets consider the case when the primary node fails given that packet processing (initial or after last packet processing progress commit) update $\mathscr{T}_j$ resulted in any Non-deterministic update in any of the NF in the chain. In such case,
Initial condition (at time of failure): $\mathscr{S}.\mathscr{V} \subseteq \mathscr{E}.\mathscr{V} \subseteq \mathscr{P}.\mathscr{V}$
On Replay condition: Now, by Proposition C.2, we have $\mathscr{E}.\mathscr{V} \subset \mathscr{P}.\mathscr{V} \ni \mathscr{E}.\mathscr{V} \bigcap \overline{\mathscr{P}.\mathscr{V}_i} = \varnothing$ cannot contain any non-deterministic update in any of the NFs. Now, again, by the Definition C.1, reprocessing of any $t_{ij} \in \mathscr{T}$ packets in $\mathbb{B}$, only updates the NF state at the secondary NF, but these packets are subsequently dropped by the Replica node and hence do not modify the $\mathscr{E}.\mathscr{V}$. However, the secondary NF(i) processes the packets, which implies: $\mathscr{S}.\mathscr{V}_i = \mathbb{P}_i * S_i \mapsto S_i'$ resulting in $\mathscr{S}.\mathscr{V}_i = \mathscr{E}.\mathscr{V}_i$
Final Status: $\mathscr{S}.\mathscr{V} = \mathscr{E}.\mathscr{V} \ni \mathscr{S}.\mathscr{V} \neq \mathscr{P}.\mathscr{V}$
Thus, again the external synchrony is preserved, but the Primary and Secondary NFs may differ in their internal states and outcome of non-deterministic packet processing.

Case 3: Lets consider the case when the packet processing results in both deterministic and Non-deterministic updates in any of the NF in the chain.

Note: $\mathbb{B}$ can contain packets that can result in non-determinism.

if $\mathscr{P}_i <= \mathscr{T} \implies$ Duplicate packet
Secondary View update but no External View.
else $\mathscr{P}_i > \mathscr{T} \implies$ New packet
No or new Secondary View update i.e. New External view.

Note: We commit state for entire NF chain *i.e.,* all the NFs in the chain get to commit the NF state during the periodic state transfer or in the event of any

non-determinism that necessitate explicit state update for any NF in the chain.

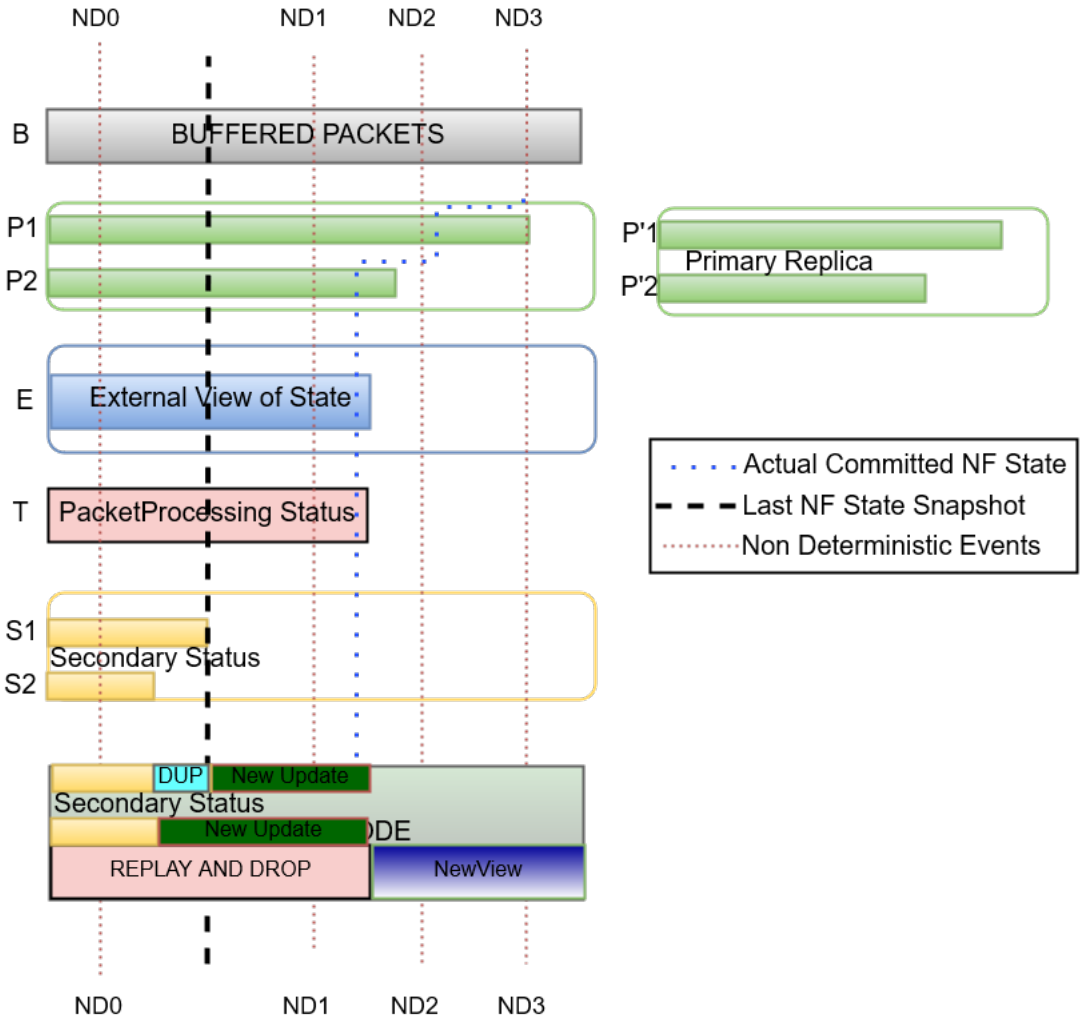


Figure C.3: Update and view of NF States across Primary, Secondary (Replica) and External observer

Figure C.3 illustrates the NF packet processing and state transfer update timelines for an NF chain with two NFs *P1* and *P2*.

Chapter D

REINFORCE Algorithms and Workflow

D.1 Work Flow Diagrams

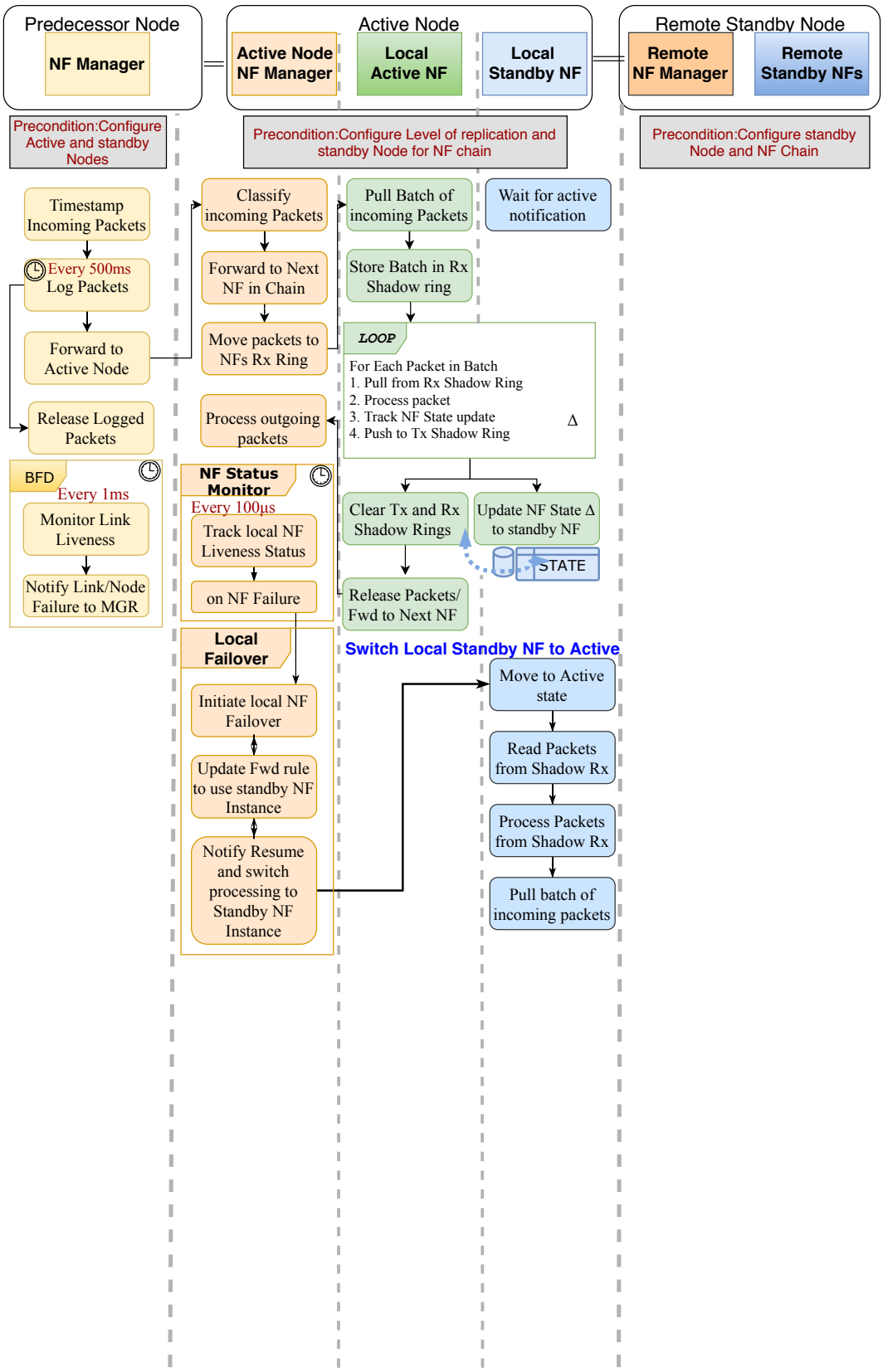


Figure D.1: Work flow for Local NF Replica and Failover scheme.

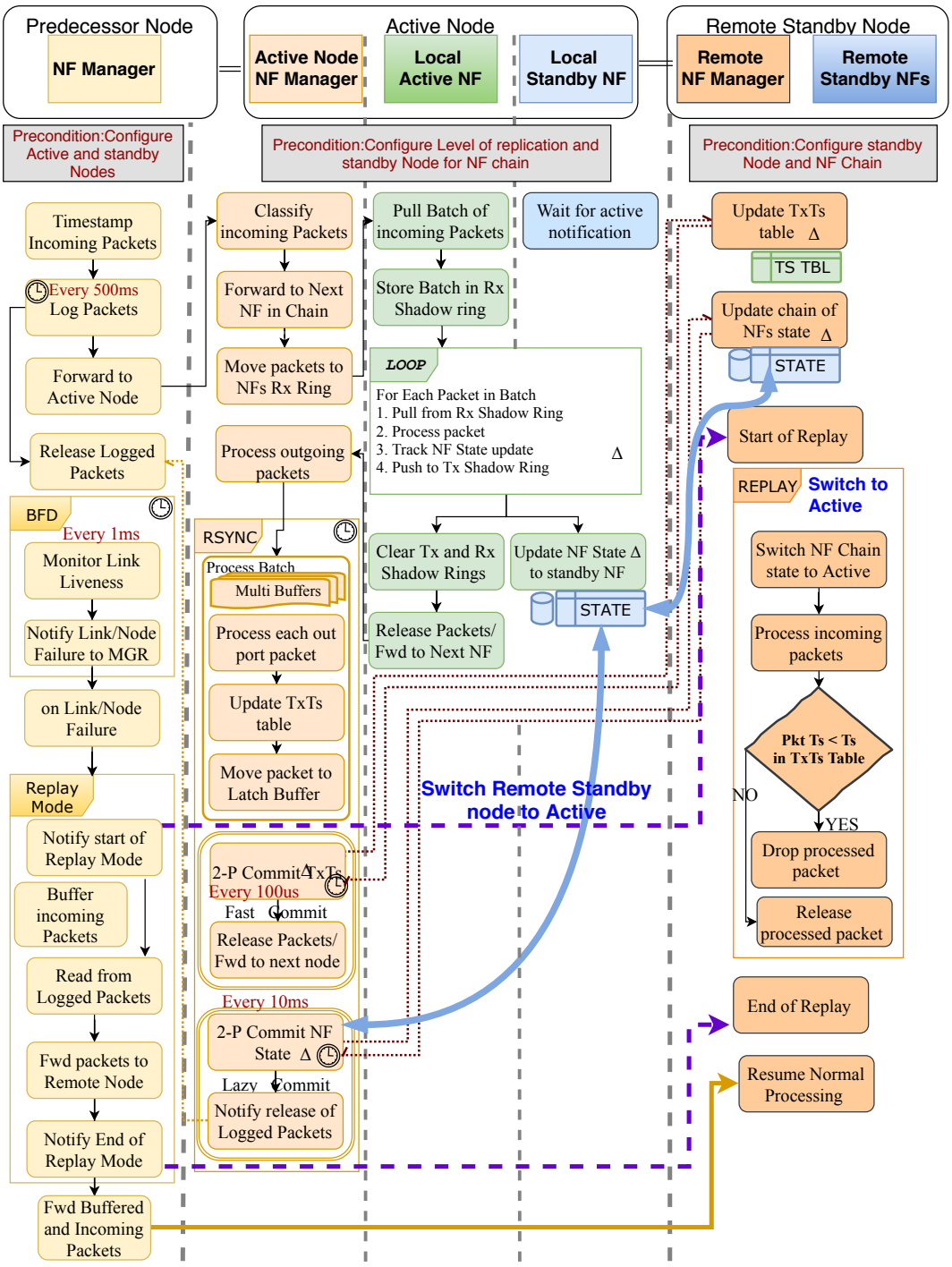


Figure D.2: Work flow for Remote NF chain Replica and Failover.

D.2 Sequence Diagram: Addressing Non-Determinism

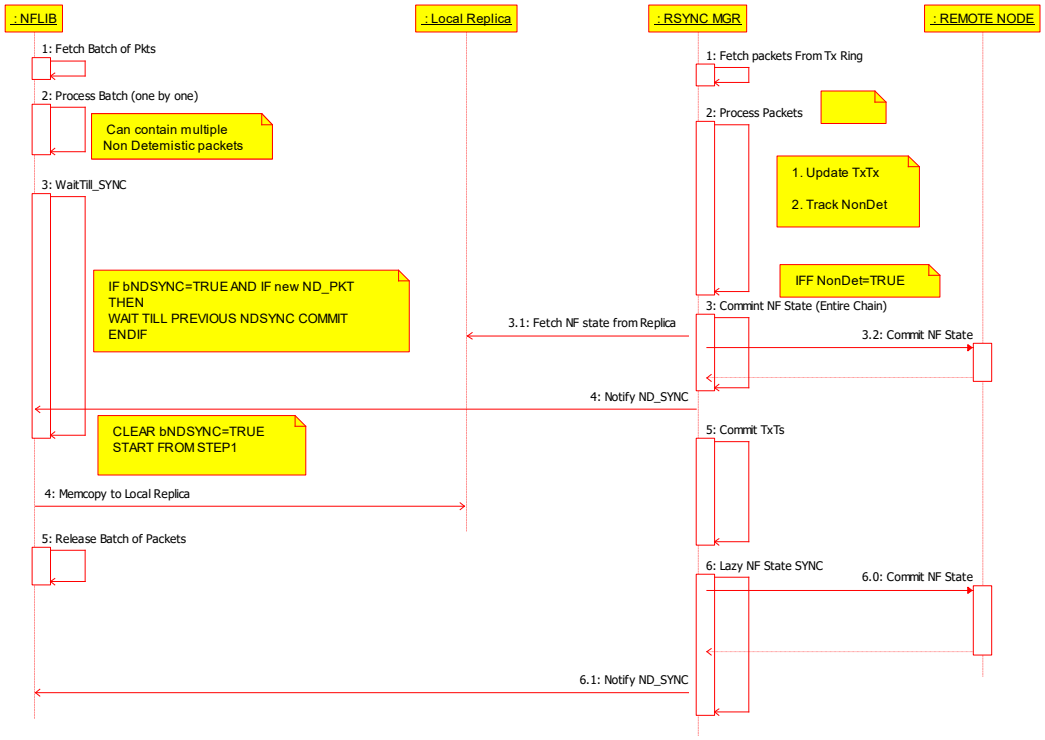


Figure D.3: Illustration of how REINFORCE addresses Non-Determinism to ensure operational correctness.

Bibiography

Bibliography

[1] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication - SIGCOMM '12*, volume 42, pages 13–24. ACM Press, 2012.

[2] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An untold story of middleboxes in cellular networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 374–385. ACM, 2011.

[3] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[4] Chunfeng Cui, Hui Deng, Deutsche Telekom, Uwe Michel, and Herbert Damker. Network functions virtualisation- introductory white paper. 2012. [ONLINE].

[5] Esti nfv isg members and participants. `https://portal.etsi.org/TBSiteMap/NFV/NFVMembership.aspx`, 2018. [ONLINE].

[6] NFVISG ETSI. NFV in ETSI. `http://www.etsi.org/technologies-clusters/technologies/nfv`, 2018. [ONLINE].

[7] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[8] Sushant Jain and et al. B4: experience with a globally-deployed software defined wan. In *ACM SIGCOMM 2013*.

[9] At&ts network built in software. `http://about.att.com/innovation/sdn`,

2018. [ONLINE].

[10] Verizon: Virtual network services. http://www.verizonenterprise.com/products/networking/sdn-nfv/virtual-network-services, 2018. [ONLINE].

[11] Axel Clauberg. Deutsche telekom terastream: A network functions virtualization (nfv) using openstack case study. [ONLINE].

[12] Caroline Chappell. Nfv mano: What's wrong & how to fix it, 2015.

[13] Paul Anderson. Virtual versus reality: The challenges of enterprise nfv adoption. https://www.sdxcentral.com/articles/contributed/the-challenges-of-enterprise-nfv-adoption/2017/10/, 2017. [ONLINE].

[14] James Crawshaw. The challenges of operationalizing nfv. http://www.lightreading.com/nfv/nfv-mano/the-challenges-of-operationalizing-nfv/a/d-id/738571, 2017. [ONLINE].

[15] NFVISG ETSI. Network Functions Virtualization (NFV): Architectural Framework. http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf, 2013. [ONLINE].

[16] Thomas Nadeau and Paul Quinn. Problem Statement for Service Function Chaining. RFC 7498, April 2015.

[17] Jrgen Quittek et al. Network functions virtualisation (nfv)-management and orchestration. *ETSI NFV ISG, White Paper*, 2014. [ONLINE].

[18] NFVISG ETSI. Network Functions Virtualization (NFV): Resiliency Requirements. http://www.etsi.org/deliver/etsi_gs/NFV-REL/001_099/001/01.01.01_60/gs_NFV-REL001v010101p.pdf, 2015. [ONLINE].

[19] NFVISG ETSI. Network Functions Virtualization (NFV): Security; Security Specification for MANO Components and Reference points . http://www.etsi.org/deliver/etsi_gs/NFV-SEC/001_099/014/03.01.01_60/gs_NFV-SEC014v030101p.pdf, 2018-04. [ONLINE].

[20] NFVISG ETSI. Network Functions Virtualization (NFV): Security; System architecture specification for execution of sensitive NFV components. https://docbox.etsi.org/Workshop/2017/201706_SECURITYWEEK/05_NFVSECURITY/

NFV-SEC012v030101p-gs-ArchitectureforsensitiveNFVcomponents.pdf, 2017. [ONLINE].

[21] J Halpern and C Pignataro. Rfc 7665: Service function chaining (sfc) architecture. https://tools.ietf.org/html/rfc7665, 2015. [ONLINE].

[22] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2. In *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM Press.

[23] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM - SIGCOMM '13*. ACM Press, 2013.

[24] Ali Mohammadkhan, Sheida Ghapani, Guyue Liu, Wei Zhang, K. K. Ramakrishnan, and Timothy Wood. Virtual function placement and traffic steering in flexible and dynamic software defined networks. In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, pages 1–6. IEEE, apr 2015.

[25] Ying Zhang, Neda Beheshti, Ludovic Beliveau, Geoffrey Lefebvre, Ravi Manghirmalani, Ramesh Mishra, Ritun Patneyt, Meral Shirazipour, Ramesh Subrahmaniam, Catherine Truchan, and Mallik Tatipamula. StEERING: A software-defined networking for inline service chaining. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, oct 2013.

[26] Clicking clean:a guide to building the green internet, 2015. Accessed: 2016-10-02.

[27] Global trends in renewable energy investment 2016. United Nations Environment Programme, Bloomberg New Energy Finance. Accessed: 2016-11-10.

[28] Markos Anastasopoulos, Anna Tzanakaki, and Dimitra Simeonidou. Stochastic energy efficient cloud service provisioning deploying renewable energy sources. *IEEE Journal on Selected Areas in Communications*, 34(12):3927–3940, dec 2016. Special Issue on Green Communications and Networking.

[29] Rahul Singh, David Irwin, Prashant Shenoy, and K.K. Ramakrishnan. Yank: Enabling green data centers to pull the plug. In *USENIX, NSDI*, 2013.

[30] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 71–84, New York, NY, USA, 2017. ACM.

[31] Argyrios G. Tasiopoulos, Sameer G. Kulkarni, Mayutan Arumaithurai, Ioannis Psaras, K. K. Ramakrishnan, Xiaorning Fu, and George Pavlou. DRENCH: A semi-distributed resource management framework for NFV based service function chaining. In *2017 IFIP Networking Conference (IFIP Networking)*. IEEE, June 2017.

[32] Sameer Kulkarni, Mayutan Arumaithurai, K. K. Ramakrishnan, and Xiaoming Fu. Neo-NSH: Towards scalable and efficient dynamic service function chaining of elastic network functions. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE, March 2017.

[33] Paul Quinn, Uri Elzur, and Carlos Pignataro. Network Service Header (NSH). RFC 8300, January 2018.

[34] Sameer G Kulkarni, Mayutan Arumaithurai, K.K. Ramakrishnan, and Xiaoming Fu. REARM: Renewable energy based resilient deployment of virtual network functions. In *2017 European Conference on Networks and Communications (EuCNC)*. IEEE, June 2017.

[35] Addressing nfv resiliency. `https://github.com/sameergk/Addressng-NFV-Resiliency`, 2018. [ONLINE].

[36] Harvey Freeman and Raouf Boutaba. Networking industry transformation through softwarization [the president's page]. *IEEE Communications Magazine*, 54(8):4–6, 2016.

[37] S. G. Kulkarni, M. Arumaithurai, A. Tasiopoulos, Y. Psaras, K. K. Ramakrishnan, Xiaoming Fu, and G. Pavlou. Name enhanced sdn framework for service function chaining of elastic network functions. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 45–46, San Francisco, USA, April 2016.

[38] NFVISG ETSI. Network Functions Virtualization (NFV): Ecosystem; Report on SDN Usage in NFV Architectural Framework. `http://www.etsi.org/deliver/etsi_gs/NFV-EVE/001_099/005/01.01.01_60/gs_NFV-EVE005v010101p.pdf`, 2015. [ONLINE].

[39] A. Kapadia and N. Chase. *Understanding Opnfv: Accelerate Nfv Transformation Using Opnfv*. CreateSpace Independent Publishing Platform, 2017.

[40] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Steven Latre, Marinos Charalambides, and Diego Lopez. Management and orchestration challenges in network functions virtualization. *IEEE Communications Magazine*, 54(1):98–105, 2016.

[41] J. Halpern and C. Pignataro. Service function chaining (sfc) architecture. RFC 7665, RFC Editor, October 2015.

[42] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, aug 1997.

[43] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, mar 2015.

[44] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, Berkeley, CA, 2012. USENIX.

[45] Data plane development kit. `http://dpdk.org/`, 2014. [ONLINE].

[46] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 26–31, New York, NY, USA, 2016. ACM.

[47] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, NSDI'14, pages 459–473, Seattle, WA, April 2014. USENIX Association.

[48] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 203–216, Berkeley, CA, USA, 2016. USENIX Association.

[49] Kenichi Yasukata, Felipe Huici, Vincenzo Maffione, Giuseppe Lettieri, and

Michio Honda. Hypernf: building a high performance, high utilization and fair nfv platform. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 157–169. ACM, 2017.

[50] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[51] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21. IEEE, IEEE, oct 1978.

[52] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[53] L. Zhang. Virtual clock: a new traffic control algorithm for packet switching networks. *ACM SIGCOMM Computer Communication Review*, 20(4):19–29, aug 1990.

[54] Ingo Molnar. Linux kernel documentation: Cfs scheduler design. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`, 2017. [ONLINE].

[55] cgroups-linux control groups. `http://man7.org/linux/man-pages/man7/cgroups.7.html`, 2017. [ONLINE].

[56] Paul Menage. Linux kernel documentation: Cgroups. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`, 2017. [ONLINE].

[57] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[58] Rahul Upadhyaya, CB Anantha Padmanabhan, Meenakshi Sundaram Lakshmanan, and Satya Routray. Optimising nfv service chains on openstack using docker. `https://www.openstack.org/videos/video/optimising-nfv-service-chains-on-openstack-using-docker`, April 2016.

[59] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks-the multiple node case. In *IEEE INFOCOM '93 The Conference on Computer Communications, Proceedings*, volume 2, pages 137–150. IEEE Comnput. Soc. Press, 1994.

[60] D. Stiliadis and A. Varma. Rate-proportional servers: a design methodology for fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 6(2):164–174, apr 1998.

[61] Cesar Marcondes and Christian Esteve Rothenberg. Tutorial: Network functions virtualization - perspectives, reality and challenges. `http://www.dca.fee.unicamp.br/~chesteve/ppt/NFV-Full-IM15-Final-Screen-v150412.pdf`, 2015. [ONLINE].

[62] Tom Kelly, Sally Floyd, and Scott Shenker. Patterns of congestion collapse. *International Computer Science Institute, and University of Cambridge*, 2003.

[63] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[64] D. Lapsley and S. Low. Random early marking: an optimisation approach to internet congestion control. In *IEEE International Conference on Networks. ICON '99 Proceedings (Cat. No.PR00243)*, pages 67–74. IEEE Comput. Soc, Sept 1999.

[65] Wu chang Feng, K.G. Shin, D.D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Transactions on Networking*, 10(4):513–528, aug 2002.

[66] I. Stoica, S. Shenker, and Hui Zhang. Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Transactions on Networking*, 11(1):33–46, feb 2003.

[67] K.K. Ramakrishnan, S. Floyd, and D. Black. RFC 3168: The Addition of Explicit Congestion Notification (ECN) to IP. `https://tools.ietf.org/html/rfc3168`, 2001. [ONLINE].

[68] VPP. `https://fd.io/`, 2016. [ONLINE].

[69] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[70] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the

cloud. Technical report, Technical Report, 2013.

[71] Dilip A. Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication - SIGCOMM '08*, volume 38, pages 51–62. ACM, ACM Press, 2008.

[72] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, nsdi'13, pages 227–240, Lombard, IL, 2013. USENIX, USENIX.

[73] Yang Li, Linh Thi Xuan Phan, and Boon Thau Loo. Network functions virtualization with soft real-time guarantees. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, IEEE, apr 2016.

[74] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM, 2013.

[75] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. Netfpga: reusable router architecture for experimental research. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7. ACM, 2008.

[76] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[77] Xiaodong Yi, Jingpu Duan, and Chuan Wu. Gpunfv: a gpu-accelerated nfv system. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 85–91. ACM, 2017.

[78] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28. ACM, 2016.

[79] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, jun 1996.

[80] P. Goyal, H.M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, 1997.

[81] D. Stiliadis and A. Varma. Efficient fair queueing algorithms for packet-switched networks. *IEEE/ACM Transactions on Networking*, 6(2):175–185, apr 1998.

[82] Luigi Rizzo, Paolo Valente, Giuseppe Lettieri, and Vincenzo Maffione. PSPAT: software packet scheduling at hardware speed. `http://info.iet.unipi.it/~luigi/papers/20160921-pspat.pdf`, 2016. [ONLINE].

[83] Anirudh Sivaraman, Nick McKeown, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, and Sachin Katti. Programmable packet scheduling at line rate. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16*, pages 44–57. ACM, ACM Press, 2016.

[84] Fibers. `https://msdn.microsoft.com/en-us/library/ms682661.aspx`, 2017. [ONLINE].

[85] Dpdk l-thread subsystem. `http://dpdk.org/doc/guides/sample_app_ug/performance_thread.html`, 2014. [ONLINE].

[86] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM - SIGCOMM '10*, volume 40, pages 63–74. ACM, ACM Press, 2010.

[87] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16*, pages 244–257. ACM, ACM Press, 2016.

[88] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ecn in multi-service multi-queue data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 537–549, Santa Clara, CA, 2016. USENIX Association.

[89] Glenn Judd. Attaining the promise and avoiding the pitfalls of tcp in the datacenter. In *NSDI*, pages 145–157, 2015.

[90] Radhika Mittal, David Zats, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, and David Wetherall. TIMELY. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication - SIGCOMM '15*, volume 45, pages 537–550. ACM, ACM Press, 2015.

[91] Changhyun Lee, Chunjong Park, Keon Jang, Sue B Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *USENIX Annual Technical Conference*, pages 403–415, 2015.

[92] Jon CR Bennett and Hui Zhang. Wf/sup 2/q: worst-case fair weighted fair queueing. In *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 1, pages 120–128. IEEE, 1996.

[93] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *ACM Sigplan Notices*, volume 44, pages 65–74. ACM, 2009.

[94] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. *ACM SIGCOMM Computer Communication Review*, 42(4):1, sep 2012.

[95] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2dfq. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16*, SIGCOMM '16, pages 144–159, New York, NY, USA, 2016. ACM Press.

[96] Performance measurements with RDTSC. `https://www.strchr.com/performance_measurements_with_rdtsc`, June 2016. [ONLINE].

[97] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference - IMC '15*, pages 275–287. ACM, ACM Press, 2015.

[98] Robert Olsson. Pktgen the linux packet generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, volume 2, pages 11–24, 2005.

[99] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu.

iperf - the ultimate speed test tool for tcp, udp and sctp. `https://iperf.fr/`, 2014.

[100] Luigi Rizzo, Stefano Garzarella, Giuseppe Lettieri, and Vincenzo Maffione. A study of speed mismatches between communicating virtual machines. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems - ANCS '16*, ANCS '16, pages 61–67, New York, NY, USA, 2016. ACM Press.

[101] Raj Jain, Dah-Ming Chiu, and William R Hawe. *A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System.*, volume 38. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.

[102] Sam Newman. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[103] Microservices architecture in the telco cloud. `https://www.sdxcentral.com/nfv/definitions/microservices-architecture-telco-cloud`, 2017. [ONLINE].

[104] What are microservices in nfv? definition. `https://www.sdxcentral.com/nfv/definitions/microservices-in-nfv-definition`, 2017. [ONLINE].

[105] Martin Fowler and James Lewis. Microservices a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html`, 2014. [ONLINE].

[106] Wassim Haddad, Heikki Mahkonen, and Ravi Manghirmalani. Nfv platforms with mirageos unikernels. `http://unikernel.org/blog/2016/unikernel-nfv-platform`, 2016. [ONLINE].

[107] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30, 2013.

[108] Chunfeng Cui and et al. Network Functions Virtualisation. In *SDN and OpenFlow World Congress*, 2012.

[109] Aaron Gember and et al. Stratos: A Network-Aware Orchestration Layer for Virtual Middleboxes in Clouds. *Tech. Rep.*, 2013.

[110] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. Programming slick network functions. In *Proceedings of the 1st ACM SIGCOMM Sym-*

posium on Software Defined Networking Research - SOSR '15. ACM Press, 2015.

[111] Timothy Wood, K. K. Ramakrishnan, Jinho Hwang, Grace Liu, and Wei Zhang. Toward a software-based network: integrating software defined networking and network function virtualization. *IEEE Network*, 29(3):36–41, may 2015.

[112] Mohammad Al-Fares, Sivasankar Radhakrishnan, and Barath Raghavan. Hedera: Dynamic Flow Scheduling for Data Center Networks. *NSDI*, 2010.

[113] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies on - CoNEXT '11*. ACM Press, 2011.

[114] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM - SIGCOMM '13*. ACM Press, 2013.

[115] Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, and Raouf Boutaba. On orchestrating virtual network functions. In *2015 11th International Conference on Network and Service Management (CNSM)*, volume abs/1503.06377. IEEE, nov 2015.

[116] Zizhong Cao, Murali Kodialam, and T. V. Lakshman. Traffic steering in software defined networks. *ACM SIGCOMM Computer Communication Review*, 44(4):98, aug 2014.

[117] Ali Mohammadkhan, Sheida Ghapani, Guyue Liu, Wei Zhang, K. K. Ramakrishnan, and Timothy Wood. Virtual function placement and traffic steering in flexible and dynamic software defined networks. In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*. IEEE, apr 2015.

[118] Jeremias Blendin, Julius Ruckert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Position paper: Software-defined network service chaining. In *2014 Third European Workshop on Software Defined Networks*. IEEE, sep 2014.

[119] Sevil Mehraghdam, Matthias Keller, and Holger Karl. Specifying and placing chains of virtual network functions. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. IEEE, oct 2014.

[120] Zizhong Cao, Murali Kodialam, and T. V. Lakshman. Traffic steering in software defined networks. *ACM SIGCOMM Computer Communication Review*, 44(4):98, aug 2014.

[121] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF. In *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14*, volume 44 of *SIGCOMM '14*, pages 163–174, New York, NY, USA, August 2014. ACM Press.

[122] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope. *SIGCOMM 2005*, 2005.

[123] Mohammad Alizadeh, Navindra Yadav, George Varghese, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, and Rong Pan. CONGA. In *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14*. ACM Press, 2014.

[124] Linqi Guo, John Pang, and Anwar Walid. Dynamic service function chaining in SDN-enabled networks with middleboxes. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, nov 2016.

[125] Kanak Agarwal, Colin Dixon, Eric Rozner, and John Carter. Shadow MACs. In *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, HotSDN '14, pages 157–162, New York, NY, USA, 2014. ACM Press.

[126] Wanfu Ding, Wen Qi, Jianping Wang, and Biao Chen. OpenSCaaS: an open service chain as a service platform toward the integration of SDN and NFV. *IEEE Network*, 29(3):30–35, May 2015.

[127] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. FlowTags. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, pages 543–546. ACM Press, 2013.

[128] Jeremias Blendin, Julius Ruckert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Demo: Software-defined network service chaining. In *2014 Third European Workshop on Software Defined Networks*, pages 139–140. IEEE, IEEE, sep 2014.

[129] Mayutan Arumaithurai, Jiachen Chen, Edo Monticelli, Xiaoming Fu, and

Kadangode K. Ramakrishnan. Exploiting ICN for flexible management of software-defined networks. In *Proceedings of the 1st international conference on Information-centric networking - INC '14*. ACM Press, 2014.

[130] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference - IMC '09*. ACM Press, 2009.

[131] Hao Jiang and Constantinos Dovrolis. Why is the internet traffic bursty in short time scales? In *SIGMETRICS 2005*.

[132] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA. In *Proceedings of the Symposium on SDN Research - SOSR '16*. ACM Press, 2016.

[133] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *The Journal of the Operational Research Society*, 49(3):237, mar 1998.

[134] R. Cohen and G. Nakibli. On the computational complexity and effectiveness of "n-hub shortest-path routing". In *INFOCOM 2004*.

[135] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59, jan 2006.

[136] Alessio Botta, Alberto Dainotti, and Antonio Pescapé. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531–3547, oct 2012.

[137] S.Kumar, M.tufail, S.Maji, C.captari, and S.Homma. Service Function Chaining Use Cases In Data Centers. *IETF draft*, 2016.

[138] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication - SIGCOMM '15*, volume 45, pages 465–478. ACM Press, August 2015.

[139] Sangeetha Abdu Jyothi, Mo Dong, and P. Brighten Godfrey. Towards a flexible data center fabric with source routing. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research - SOSR '15*, pages 1–8. ACM Press, 2015.

[140] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle. In *Proceedings of the 2013 conference on Internet measurement conference - IMC '13*, IMC '13, pages 9–22, New York, NY, USA, 2013. ACM Press.

[141] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers. *ACM SIGCOMM Computer Communication Review*, 41(4):350, oct 2011.

[142] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing? In *Proceedings of the Seventh ACM Symposium on Cloud Computing - SoCC '16*, SoCC '16, pages 1–16, New York, NY, USA, 2016. ACM Press.

[143] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, ICSE '10, pages 485–494, New York, NY, USA, 2010. ACM Press.

[144] Haryadi S. Gunawi, Vincentius Martin, Anang D. Satria, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, and Jeffrey F. Lukman. What bugs live in the cloud? In *Proceedings of the ACM Symposium on Cloud Computing - SOCC '14*, SOCC '14, pages 7:1–7:14, New York, NY, USA, 2014. ACM Press.

[145] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels. *ACM SIGPLAN Notices*, 48(4):461, apr 2013.

[146] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, aug 2000.

[147] Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. Trends in worldwide ICT electricity consumption from 2007 to 2012. *Computer Communications*, 50:64–76, sep 2014. Green Networking.

[148] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. 2011.

[149] Green House Data, A report on Energy Consumption. Accessed: 2016-10-02.

[150] Justine Sherry, Luigi Rizzo, Scott Shenker, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, and Sylvia Ratnasamy. Rollback-recovery for middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication - SIGCOMM '15*, volume 45, pages 227–240, New York, NY, USA, August 2015. ACM Press.

[151] Chao Li, Amer Qouneh, and Tao Li. iSwitch: Coordinating and optimizing renewable energy powered server clusters. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, jun 2012.

[152] Markos P. Anastasopoulos, Anna Tzanakaki, Bijan Rahimzadeh Rofoee, Shuping Peng, Yan Yan, Dimitra Simeonidou, Giada Landi, Giacomo Bernini, Nicola Ciulli, Jordi Ferrer Riera, Eduard Escalona, Kostas Katsalis, and Thanasis Korakis. Optical wireless network convergence in support of energy-efficient mobile cloud services. *Photonic Network Communications*, 29(3):269–281, apr 2015. Special Issue on Green Communications and Networking.

[153] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication. In *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13*, SOCC '13, pages 1:1–1:15, New York, NY, USA, 2013. ACM Press.

[154] Junaid Khalid and Aditya Akella. Streamnf: Performance and correctness for stateful chained nfs. *CoRR*, abs/1612.01497, 2016.

[155] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, 2017. USENIX Association.

[156] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM Press.

[157] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 1st edition, 2016.

[158] Rashid Mijumbi and et al. On the energy efficiency prospects of network function virtualization. *CoRR*, abs/1512.00215, 2015.

[159] Raffaele Bolla, Chiara Lombardo, Roberto Bruschi, and Sergio Mangialardi.

DROPv2: energy efficiency through network function virtualization. *IEEE Network*, 28(2):26–32, mar 2014.

[160] Zhifeng Xu, Fangming Liu, Tao Wang, and Hong Xu. Demystifying the energy efficiency of network function virtualization. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*. IEEE, jun 2016.

[161] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems*, 26(3):1–26, sep 2008.

[162] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems*, 26(3):1–26, sep 2008.

[163] Criu: Checkpoint restore in userspace. `http://criu.org/`, 2017. [ONLINE].

[164] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, June 2010.

[165] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop). RFC 5881, June 2010.

[166] Dave Katz and David Ward. Generic Application of Bidirectional Forwarding Detection (BFD). RFC 5882, June 2010.

[167] Carlos Pignataro, David Ward, Nobo Akiya, Manav Bhatia, and Juniper Networks. Seamless Bidirectional Forwarding Detection (S-BFD). RFC 7880, July 2016.

[168] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[169] Luca Deri, Maurizio Martinelli, Tomasz Bujlow, and Alfredo Cardigliano. nDPI: Open-source high-speed deep packet inspection. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 617–622. IEEE, aug 2014.

[170] wrk: a http benchmarking tool. `https://github.com/wg/wrk`, 2018. [ONLINE].

[171] ndpi test pcap traces. `https://github.com/ntop/nDPI/tree/dev/tests/`

`pcap`, 2018. [ONLINE].

[172] Verizon Networks Infrastructure Planning, SDN-NFV Reference Architecture, February 2016.

[173] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, Jacobus Van der Merwe, Jinho Hwang, Guyue Liu, and Lucas Chaufournier. CloudNet: Dynamic pooling of cloud resources by live WAN migration of virtual machines. *IEEE/ACM Transactions on Networking*, 23(5):1568–1583, oct 2015.

[174] amdocs Inc. Nfv survey: Approaches to network virtualization. `http://solutions.amdocs.com/nfv-approaches.html`, 2017. [ONLINE].

[175] Nfv report series part 1: Nfv infrastructure (nfvi) and vim. `https://www.sdxcentral.com/reports/2018/nfv-infrastructure-nfvi-vim/`, 2018. [ONLINE].

[176] NFVISG ETSI. Network functions virtualisation (nfv); terminology for main concepts in nfv. *Group Specification*, 2014.

[177] Tua A. Tamba. On handelman's representation of network utility maximization. *The Computer Journal*, pages 1–10, 2017.

[178] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 24(4):361–392, 2006.

SAMEER G KULKARNI

EDUCATION

PhD Candidate, Computer Science (advisor: Prof. Xiaoming Fu) Mar 2015 - present
University of Göttingen, Göttingen, Germany
Master of Science, Computer Engineering (advisor: Prof. Kai Hwang) Aug 2008 - May 2010
University of Southern California, Los Angeles, USA **GPA: 3.76/4.0**
Bachelor of Engineering, Computer Science & Engineering Aug 2000 - June 2004
National Institute of Engineering, Mysore, India (VTU) **First Class with Distinction**

PUBLICATIONS

- **Kulkarni, S.G.**, Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K.K., Wood, T., Arumaithurai, M. and Fu, X. (2017). NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In Proceedings of the Conference of the ACM Special Interest Group on Data Communications (SIGCOMM 2017), ACM, pp. 71-84, Los Angeles, USA, August 2017.
- Tasiopoulos, A., **Kulkarni, S.G.**, Arumaithurai, M., Psaras, I., Ramakrishnan, K.K., Fu, X. and Pavlou, G. (2017). DRENCH: A Semi-Distributed Resource Management Framework for NFV based Service Function Chaining. In Proceedings of IFIP Networking 2017 Conference (Networking 2017), pp. 1-9, Stockholm, Sweden, June 2017. ***(Joint First Author)***
- **Kulkarni, S.G.**, Arumaithurai, M., Ramakrishnan, K.K. and Fu, X. (2017). REARM: Renewable energy based resilient deployment of Virtual Network Functions. In Proceedings of 26th European Conference on Networks and Communications (EuCNC), IEEE, pp. 1-6, Oulu, Finland, June 2017.
- **Kulkarni, S.G.**, Arumaithurai, M., Ramakrishnan, K.K. and Fu, X. (2017). Neo-NSH: Towards scalable and efficient dynamic service function chaining of elastic network functions. In Proceedings of 20th Conference on Innovations in Clouds, Internet and Networks (ICIN 2017), IEEE, pp. 308-312, Paris, France, March 2017.
- **Kulkarni, S.G.**, Arumaithurai, M., Tasiopoulos, A., Psaras, Y., Ramakrishnan, K.K., Fu, X. and Pavlou, G. (2016). Name enhanced sdn framework for service function chaining of elastic network functions. In Proceedings on IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS 2016), IEEE, pp. 45-46, San Francisco, CA, USA, April 2016.
- Hwang, K., **Kulkarni, S.G.** and Hu, Y. (2009). Cloud security with virtualized defense and reputation-based trust management. In Proceedings of 8th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC 2009), IEEE, pp. 717-722, Chengdu, China, December 2009.
- **Kulkarni, S.G.**, Liu, G., Ramakrishnan, K.K., Arumaithurai, M. Wood, T. and Fu, X. (2018). REINFORCE: Achieving Efficient Failure resiliency for Network Function Virtualization based Services. *Submitted to ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT 2018)* ***(under review)***.

RESEARCH EXPERIENCE

EU ITN CleanSky Early-stage Researcher, University of Göttingen. [Mar 2015 - present]

- Carrying out reasearch with foucs on Software Defined Networking and Network Function Virtualization to build a robust and reliable network for Cloud Computing eco-system.

Directed Research – University of Southern California, Los Angeles

- Addressing Security, Reliability and Trust Management for Internet Cloud [FALL 2009]

INDUSTRY EXPERIENCE

Engineer Staff – Multimedia Systems, QCT, Qualcomm, San Diego. June 2010 - Feb 2015

- Responsible for development of core sound driver, OpenMAX-IL audio middleware stack for Qualcomm Snapdragon chips. processors for BlackBerry and Android (KitKat & Ice cream Lollipop) product lines.

Sr. Engineer – Product Research & Design (Embedded Systems), Tata Elxsi, Bangalore. Aug 2004 - July 2008.

- Development of multimedia widgets for Sagem mobiles and HDTV middleware system for different broadcasting standards.

REASEARCH VISITS

- Nokia Bell Labs, Stuttgart, Germany (host: Dr. Volker Hilt), May 2017 - Aug 2017.
- Department of Computer Science and Engineering, University of California, Riverside, CA, USA (host: Prof. K. K. Ramakrishnan), Jul 2016 - Dec 2016.

TEACHING EXPERIENCE

- Teaching Assistant:
  - Practical Course Networking Lab, University of Göttingen, Apr 2015 - present.
  - Software Defined Networking course, University of Göttingen, Apr 2016 - present.
  - Advanced SDN and NFV course, University of Göttingen, Apr 2016 - Feb 2017.
- Supervised more than 8 master students in Seminar on Internet Technology, University of Göttingeen, Apr 2015 - present.
- Mentor for interns: Multimedia Audio team, Qualcomm, San Diego, CA, Jan 2013 - Feb 2015.
- CCB and Power team member: Multimedia systems API review & control board, Audio Power analysis & Optimizations, Qualcomm, San Diego, CA, Jan 2011 - Feb 2015.
- Lab coordinator: GridSec Cluster Lab, University of Southern California, Aug 2009 - May 2010.

THESIS SUPERVISION

- Martinez Gudio. *NSH Routing: Implementation of Network Service Headers to realize the service chain by steering traffic across the VNFs.* Master Thesis, July 2017 - Jan 2018.
- Hari Raghavendrarao Bhandari. *Evaluation of state of the art works on SDN/NFV Placement and Load Balancing.* Master Thesis, Dec 2016 - May 2017.

HONORS AND AWARDS

- Travel Grants
  - ACM Travel Grants, Special Interest Group on Data Communications (SIGCOMM) Conference, Student Travel Grant, Aug 2017
  - European Conference on Networking and Communications (EuCNC) Conference, Student Travel Grant, June 2017
- Seven-time recipient of QUALSTAR award at Qualcomm, San Diego for exemplary work in different aspects of Audio sub system. June 2010 – Feb 2015.

- Received special accolades from SAGEM for developing MVC based multimedia editor in C language at Tata Elxsi, Bangalore, Nov 2005.
- Best research project for "Design of Network File System on Linux" - Computer Science and Engineering Department, NIE Mysore, June 2004.
- Best "C/C++" programmer award at "Code Drills: Annual Tech festival" NIE, Mysore, Mar 2002.

COMMUNITY SERVICES

- *Journal Reviews*:
  - IEEE Communications Magazine
  - IEEE Transactions on Network and Service Management
  - Elsevier Computer Communications (COMCOM)
  - Elsevier Computer Standards & Interfaces (CSI)
- *Conference Paper Reviews*:
  - 2018: ACM SIGMETRICS, IFIP Networking, IEEE ICC
  - 2016 & 2015: IEEE/ACM IWQoS
- Conference Organization and Volunteering
  - Volunteer in the organizing committee for NetSys 2017 Conference, Göttingen.
  - Student volunteer for ACM CoNEXT 2016 conference, Irvine, CA, USA.
- Organized "Codewars: Software programming contest" and "Conundrum Hour: Puzzles & Brain Teasers", Technical events of IEEE Student Branch at NIE, Mysore, May 2004.