LEIBNIZ UNIVERSITÄT HANNOVER
FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK

Software-Defined Middlebox Networking

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades

Doktor-Ingenieur

genehmigte

Dissertation

von

M.Sc., Ahmed Mohamed Ahmed Abujoda
geboren am 15 February 1983 in Elkamlin, Sudan

2016

REFERENT	:	Prof. Dr. Panagiotis Papdimitriou
KORREFERENT	:	Prof. Dr. rer. nat. Torsten Braun
TAG DER PROMOTION	:	10.02.2016

M.Sc., Ahmed Mohamed Ahmed Abujoda : *Software-Defined Middlebox Networking*, Dissertation, © 10.02.2016

ABSTRACT

Middleboxes have become an essential part of the Internet providing a wide range of crucial functions such as encryption, caching and redundancy elimination. However, despite their important role, today's middleboxes are single-role devices built of specialized hardware, provisioned for peak load and managed through device and vendor specific interfaces leading to inefficient resource management and high capital and operation cost. Network Function Virtualization (NFV) is a new concept aiming at mitigating the limitations of middleboxes by outsourcing Network Functions (NFs) to virtualized infrastructures and enabling new bussiness models, e.g., Network Function as a Service (NFaaS). Yet, while bringing significant advantages to the network, NFV raises serious challenges in terms of network service deployment, whether this deployment takes place on the traffic path (to avoid redirection and hence latency inflation) or off the path (to support NFs with high processing demand).

The deployment of NFs by Network Function Providers (NFPs) (i.e., ISPs providing virtualized infrastructure for NFs deployment) should maintain a certain order (i.e., service chain) to enforce the enterprise policy, while taking into account the location-dependency of some NFs. The latter might raise the need for collaboration between different NFPs to deploy NFs, given the limited geographical footprint of a single NFP. This, in turn, entails serious challenges in terms of the privacy and the autonomy of each NFP, i.e., NFPs are known to implement different local policies while being secretive about their network resources and topology. To this end, we need to develop multi-provider approaches for NFs assignment (i.e., mapping NFs to processing and network resources) and middlebox discovery (for on-path processing), while preserving the autonomy and the privacy of each provider and maintaining the NFs order. Furthermore, as with any service offering, these approaches should also enable competitive price offering of network services to clients.

In this thesis, we present MIDAS, an architecture for the coordination of middlebox discovery and selection across multiple NFPs. MIDAS relies on a centralized middlebox controller in each NFP to provide interoperability among NFPs for flow processing setup. MIDAS establishes on-path processing via middlebox signaling, controller chaining, and Multi-Party Computation (MPC) based middlebox selection. We particularly employ MPC to preserve the confidentiality of middlebox utilization across the NFPs. We study the feasibility of MIDAS using a prototype implementation and further present simulation results to assess the efficiency of our middlebox selection

approach. Our results show that MIDAS incurs low setup delay in the order of tenths of milliseconds while achieving network-wide load balancing and high request acceptance rate.

We also propose DistNSE, a distributed architecture that enables the collaboration among NFPs for off-path Network Service Embedding (NSE) (i.e., the mapping of service chains), while maintaining the privacy and the autonomy of each NFP. DistNSE also ensures competitive pricing by letting different NFPs compete for different NFs of a service chain. To this end, DistNSE decomposes NSE in two steps: (i) inter-provider embedding, where we propose an algorithm in order to partition a service chain and establish competition across NFPs, and (ii) intra-provider embedding, where we allow NFPs to enforce different policies for the assignment of chain segments to datacenter networks. We further couple the proposed embedding methods with a communication protocol for interoperability and collaboration among the participating NFPs. We use simulations to assess the efficiency of DistNSE and identify significant gains over an existing distributed embedding framework (i.e., Polyvine) in terms of service and embedding cost.

Both DistNSE and MIDAS require the knowledge of NFs computational requirements to assign NFs to processing platforms (e.g., servers). Hence, in this thesis we investigate the implications and challenges arising from NFs' workload profiling on commodity servers. We exemplify a technique that circumvents the difficulty of profiling packet processing workloads. Applying this technique to our packet processing platform, we gauge the computational requirements of selected workloads and corroborate the effect of various I/O optimizations on workload CPU utilization.

Keywords: Network Function Virtualization, Software Defined Networking, Middlebox.

ZUSAMMENFASSUNG

Middleboxes sind heute ein wesentlicher Teil des Internets und bieten eine breite Palette von wichtigen Funktionen wie Verschlüsselung, Caching und Reduktion von Redundanzen. Doch trotz ihrer wichtigen Rolle sind heutige Middleboxen hochspezialisierte Geräte, die eine einzige Netzwerkfunktion beherrschen. Desweiteren sind diese Gerate für Spitzenlasten dimensioniert und besitzen in der Regel Herstellerspezifische Schnittstellen, und führen somit zu einem ineffizienten Ressourcenmanagement mit hohen Kapital- und Betriebskosten.

Network Function Virtualization (NFV) ist ein neues Konzept, welches das Ziel hat, die Einschränkungen von Middleboxen durch Auslagerung von Netzwerkfunktionen (NFs) auf virtualisierte Infrastrukturen aufzuheben und somit neue Geschäftsmodelle, wie z.B. Network Functions as a Service (NFaaS) zu ermöglichen. Doch trotz der erheblichen Vorteile bringt das NFV-Konzept auch große Herausforderungen für die Verwendung von Netzwerkfunktionen mit sich, sowohl für deren Einsatz entlang eines Netzwerkpfades (on-path), wo zusätzliche Verzögerungen vermieden werden müssen, als auch abseits des Pfades, um NFs mit hohen Rechenanforderungen zu Unterstützen.

Der Platzierung von NFs bei Network Function Providern (NFPn) - ISPs die eine virtualisierte Infrastruktur für die Bereitstellung von NFs anbieten - muss die korrekte Reihenfolge der NFs in einer sogennanten Service-Chain und deren geografische Abhängigkeit sicherstellen. Letzteres erfordert die Zusammenarbeit zwischen den verschiedenen NFPs, da die geographische Reichweite eines einzelnen NFP begrenzt ist. Dies wiederum bringt große Herausforderungen in Bezug auf den Schutz der Privatsphäre und der Autonomie der einzelnen NFP: in der Regel implementieren NFPs heute unterschiedliche regulatorische Richtlinien und machen ihre Netzwerkressourcen sowie die verwendete Topologie nicht öffentlich. Daher müssen Multi-Provider-Ansätze entwickelt werden, welche sowohl eine Abbildung von NFs auf Netzwerkressourcen ermöglichen, als auch das Discovery von Middleboxen erlauben und dabei die Autonomie und Privatsphäre der einzelnen Anbieter aufrechterhalten. Darüber hinaus, wie bei jedem Service-Angebot, sollten diese Ansätze den NFPs ermöglichen, wettbewerbsfähige Preise für Netzwerk-Dienstleistungen zu erzielen.

In dieser Arbeit präsentieren wir MIDAS, eine Architektur zur Koordinierung der Middlebox Discovery und Auswahl über mehrere NFPs. MIDAS benutzt hierzu einen zentralisierten Controller in jedem NFP, um die Interoperabilität zwischen NFPs für den Aufbau des Flowprocessing bereitzustellen. MIDAS ermöglicht on-path Ver-

arbeitung mittels Middlebox-Signalisierung, Controller-Chaining und Middlebox Auswahl mit Hilfe von Multi-Party Computation (MPC). Insbesondere setzen wir MPC ein, um die Vertraulichkeit der Middlebox-Auslastung über mehrere NFPs sicherzustellen. Wir untersuchen die Machbarkeit des MIDAS Architektur mithilfe einer prototypische Implementation und präsentieren Simulationsergebnisse, die die Effizienz unseres Auswahlansatzes bewerten. Unsere Ergebnisse zeigen, dass MIDAS eine Aufbauverzögerung in der Größenordnung von Zehntel Millisekunden hat, während gleichzeitig ein netzwerkweiter Lastverteilung und eine hohe Anforderungsannahmequote erreicht wird.

Zusätzlich präsentieren wir DistNSE, ein verteiltes Framework, welches die Zusammenarbeit zwischen NFPs für off-path Network Service Embedding (NSE) ermöglicht, d.h., die Zuordnung von Service-Chains, unter Beibehaltung der Privatsphäre und Autonomie der einzelnen NFP. DistNSE stellt ebenfalls eine wettbewerbsfähige Preisgestaltung sicher, indem verschiedene NFPs um verschiedene NFs einer Service-Chain konkurrieren. Zu diesem Zweck analysiert DistNSE das NSE Problem in zwei Schritten: (i) das Inter-Provider-Embedding, wofür wir einen Algorithmus entwickeln, der die Service-Chain partitioniert und Wettbewerb zwischen den NFPs sicherstellt, und (ii) Intra-Provider-Embedding, wo wir NFPs ermöglichen unterschiedliche Richtlinien für die Vergabe von Rechenzentrumsnetzwerken einzusetzen. Weiterhin entwickeln wir ein Kommunikationsprotokoll für die Koordinierung zwischen den NFPs. Wir verwenden Simulationen, um die Effizienz von DistNSE zu bewerten und identifizieren signifikante Vorteile gegenüber einem bestehenden verteilten Einbettungsalgorithmus (Polyvine) in Bezug auf Service und Einbettungskosten.

Sowohl DistNSE als auch MIDAS erfordern die Kenntnis der Rechenanforderungen der NFs, um diese NFs den Verarbeitungsplattformen (z.B. Server) zuzuweisen. Daher wird in dieser Arbeit untersucht, welche Auswirkungen und Herausforderungen aus NFs Workload-Profilierung auf Commodity-Servern entsteht. Wir entwicklen anschließend eine Technik, welche die Schwierigkeit des Profilings von Paketverabeitungsoperationen umgeht. Diese Technik kommt in unserer Paket-Verarbeitungsplattform zum Einsatz, in der wir die Rechenanforderungen von ausgewählten Workloads messen und die Wirkung von verschiedenen I/O-Optimierungen auf die Arbeitsbelastung der CPU-Auslastung bestätigen konnten.

Schlagwörter: Network Function Virtualization, Software Defined Networking, Middlebox.

MY PUBLICATIONS

Parts of this thesis are based on the following papers that have already been published.

A. Abujoda and P. Papadimitriou. Profiling Packet Processing Workloads on Commodity Servers. In *Wired/Wireless Internet Communication*, volume 7889 of *Lecture Notes in Computer Science*, pages 216–228. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38400-4. doi: 10.1007/978-3-642-38401-1_17. URL http://dx.doi.org/10.1007/978-3-642-38401-1_17.

A. Abujoda and P. Papadimitriou. MIDAS: Middlebox discovery and selection for on-path flow processing. In *Communication Systems and Networks (COMSNETS), 2015 7th International Conference on*, pages 1–8, Jan 2015a. doi: 10.1109/COMSNETS.2015.7098686.

A. Abujoda and P. Papadimitriou. Invariant Preserving Middlebox Traversal. In *Wired/Wireless Internet Communications*, volume 9071 of *Lecture Notes in Computer Science*, pages 139–150. Springer International Publishing, 2015b. ISBN 978-3-319-22571-5. doi: 10.1007/978-3-319-22572-2_10. URL http://dx.doi.org/10.1007/978-3-319-22572-2_10.

A. Abujoda and P. Papadimitriou. DistNSE: Distributed Network Service Embedding Across Multiple Providers. In *Communication Systems and Networks (COMSNETS), 2016 8th International Conference on*, Jan 2016.

A. Abujoda, A. Sathiaseelan, A. Rizk, and P. Papadimitriou. Software-defined crowd-shared wireless mesh networks. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2014 IEEE 10th International Conference on*, pages 130–135, Oct 2014. doi: 10.1109/WiMOB.2014.6962161.

A. Abujoda, D. Dietrich, P. Papadimitriou, and A. Sathiaseelan. Software-Defined Wireless Mesh Networks for Internet Access Sharing. *Computer Networks*, 2015. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/j.comnet.2015.09.008. URL http://www.sciencedirect.com/science/article/pii/S1389128615003187.

D. Dietrich, A. Abujoda, and P. Papadimitriou. Network service embedding across multiple providers with nestor. In *IFIP Networking Conference (IFIP Networking), 2015*, pages 1–9, May 2015. doi: 10.1109/IFIPNetworking.2015.7145312.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AES Advanced Encryption Standard

API Application Programming Interface

AS Autonomous System

BGP Border Gateway Protocol

CLI Command Language Interpreter

CM Control Module

CoMB Consolidated Middleboxes

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DC Datacenter

DistNSE Distributed Network Service Embedding

DMA Direct Memory Access

DNS Domain Name System

DPDK Data Plane Development Kit

ECN Explicit Congestion Notification

GPU Graphics Processor Unit

HSA Header Space Analysis

HTTP Hypertext Transfer Protocol

IDS Intrusion Detection System

IP Internet Protocol

IPS Intrusion Prevention System

ISP Internet Service Provider

MAC Media Access Control

MIDAS Middlebox Discovery and Selection for On-Path Flow Processing

MPC Multi-Party Computation

NAT Network Address Translator

NetConf Network Configuration Protocol

NFaaS Network Function as a Service

NF Network Function

NFP Network Function Provider

NFV Network Function Virtualization

NFVI NFV Infrastructure

NIC Network Interface Card

NPCL Network Processing Client

NSE Network Service Embedding

NSIS Next Steps In Signaling

NUMA Non-Uniform Memory Access

OS Operating System

OVSDB Open vSwitch Database

PCIe Peripheral Component Interconnect express

PM Processing Module

P2P Peer to Peer

PoPs Points-of-Presence

PSM Packet Steering Module

QoS Quality of Service

RAO Router Alert Option

RE Redundancy Elimination

RSPAN Remote Switched Port Analyzer

RSVP Resource Reservation Protocol

SCTP Stream Control Transmission Protocol

SDN Software-Defined Networking

SIMCO Simple Middlebox Configuration

SRID Service Request ID

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TTL Time To Live

UDP User Datagram Protocol

VM Virtual Machine

VoIP Voice over IP

WAF Web Application Firewall

WAN Wide Area Network

Part I

DISSERTATION

1 INTRODUCTION

The Internet has become an integral part of our daily life providing services and applications for research, education, health care, entertainment, politics and trading. Today, the Internet serves more than 3.2 billion users (almost half the world population) [15] and generates more than $2.3 trillion in revenue worldwide [73]. In fact, the Internet has become so essential that the United Nations has declared Internet access as a human right [117].

The Internet owes much of its success to its original design. In principle, the Internet was designed to provide a simple and transparent network and rely on end nodes for application-specific functions and intelligent processing (known also as the end-to-end principle) [50]. Based on this design, the network delivers packets unmodified (except the routing and forwarding fields such as Time To Live (TTL) and MAC addresses) from one node to the other, whereas any processing or modification on the packet content is performed by the end nodes. This design principle allowed the Internet to be extensible and general, accommodating emerging protocols and applications that could have never been anticipated at its early days. However, as it continued to grow, the Internet faced new requirements that could not be fulfilled by the original design. Specifically, the Internet had to deal with challenges such as protecting the network and the end users against the increasing number of attacks and threats, providing delay and bandwidth guarantee for new applications (e.g., VoIP, IPTV, gaming), improving performance and minimizing cost (e.g., conserve access bandwidth and minimize delay), scaling beyond the limited space of the IPv4 address, and enhancing monitoring and control. As a result, network administrators and Internet Service Providers (ISPs) started deploying new network appliances, known as middleboxes, on their networks [122][130]. Middleboxes augment the Internet with a wide range of Network Functions (NFs) such as packet filtering, intrusion detection, encryption, load balancing, caching, and redundancy elimination. Over the years, middleboxes have become indispensable for the network functionality. Indeed, a recent study [122] found that the number of middleboxes in enterprise networks is as high as the number of routers and switches.

However, despite their important role, today's middleboxes suffer from a set of limitations. They are built of dedicated hardware boxes and serve a specific purpose. Hence, in order to deploy new functions or upgrade the existing ones, network administrators are required to acquire new devices which leads to high investment costs [122]. Fur-

thermore, the fact that middleboxes are standalone devices that are provisioned for peak loads implies that most of the time they are underutilized (or overutilized, when traffic load suddenly increases) with no opportunity for consolidation or resources pooling within and across different middleboxes. In terms of management and configuration, the diverse and complex functions of middleboxes require a wide range of technical expertise and skills which may vary across various vendors offering the same middlebox function. This, in turn, leads to a large increase in operation cost incurred through training cost or through hiring skilled administrators [122].

Network Function Virtualization (NFV) is a recent trend aiming at mitigating these limitations by replacing the special purpose hardware-based middleboxes with software-based NFs running on virtualized generic compute platforms [13, 34, 35, 22]. More specifically, NFV enables the consolidation of software NFs on standard commodity servers [64], switches and storage which could be placed in the ISP network or in remote datacenters. As with cloud computing, outsourcing NFs to datacenters opens the door for new service models such as Network Function as a Service (NFaaS) [13]. NFaaS enables network operators to rent their compute resources to host clients' NFs in a pay-per-use fashion leading to operation and investment cost saving and dynamic on-demand-based resource provisioning. Furthermore, NFV enables network processing which can facilitate the deployment of new services and is essentially a prerequisite for emerging communication paradigms, such as information-centric networking [93]. Packet inspection and filtering, redundancy elimination, and caching in the network can provide better support for many applications, especially when such NFs are optimized for certain applications. Network processing can also result in bandwidth conservation. For example, the deployment of intrusion detection systems and packet filters in the network can conserve bandwidth by filtering suspicious flows upstream near their source.

Inline with NFV, ISPs, such as AT&T, Deutsche Telekom, and Telefonica, have started the deployment of small datacenters (known as micro-datacenters) to enhance the support for content distribution [78] and offer virtualized NFs to their clients. These datacenters are placed at Points-of-Presence (PoPs) providing a wide geographical footprint and low latency. In these datacenters, commodity servers will be first-class citizens for hosting NFs, due to their unprecedented level of programmability and flexibility [119, 64, 81]. Furthermore, with CPUs with instruction sets optimized for certain NFs, GPUs with massive level of parallelism [86], and recent technologies for high-speed packet I/O (e.g., netmap [115], DPDK [1]), commodity servers will perform a multitude of packet processing operations at high rates for NFs deployment.

However, while bringing significant benefits to the network, NFV raises serious requirements in terms of network service deployment. In particular, NFV requires the development of techniques and approaches for resource discovery and allocation to place NFs on the networks and datacenters of the Network Function Providers (NFPs) (i.e., ISPs providing virtualized infrastructure for NFs deployment). Typically, NFs can be deployed either on or off the traffic path [122] [132]. On-path processing obviates the need for traffic redirection which can lead to latency inflation and high bandwidth consumption. However, on-path processing also requires the presence of processing platforms with sufficient available resources along the traffic path. Furthermore, all traffic is forced to pass through the on-path middleboxes which may involve unwanted processing on the traffic and introduce a single of point of failure. On the other hand, off-path processing provides plenty of resources by redirecting traffic to datacenters deployed off the traffic path. It also avoids unwanted processing and possible disruption due to middlebox failure. Yet, to reach datacenters deployed off the path, the traffic may have to traverse longer paths leading to latency inflation and high bandwidth consumption. Placing network functions on or off the path depends on the provider's policy and available resources (e.g., the availability of processing platforms on the path), and the client requirements (e.g., delay constrains). For example, a client aiming to transcode or compress a video stream might prefer on-path processing due the low delay, whereas a client who wants to encrypt delay-tolerant application traffic (e.g., file transfer) might request off-path processing due to the amount of processing capacity available in datacenters, since encryption has high computational requirements.

To this end, the main objective of this thesis is to enable network service deployment on and off the traffic path. In particular, we investigate and develop different techniques for resource discovery and allocation for NFV across and within NFPs. In the following sections, we discuss the main challenges for network service deployment, outline our approaches and contributions, and present the structure of this thesis.

1.1 CHALLENGES AND REQUIREMENTS

In the following, we discuss the main challenges for network service deployment:

- **Correctness:** Middleboxes are typically deployed in a particular order to fulfil certain policies, e.g., traffic should traverse a firewall to filter suspicious traffic before reaching a cache. In enterprise networks, such policies are enforced by the careful placement of middleboxes in the network. The deployment of middleboxes should preserve the order specified by the enter-

prise policy. Ideally, the migrated middleboxes should enforce policies identical to middleboxes deployed on-site.

- **Location dependency:** Some NFs are location dependant, e.g., cache and redundancy elimination should be deployed close to the client side to minimize delay and conserve bandwidth, whereas the load balancer and web application firewall should be deployed in proximity to the server side to distribute traffic load and prevent malicious access. Since a single NFP may not satisfy the location constraints of all NFs in a service chain (i.e., a sequence of NFs commonly specified as a service chain) due to its limited geographical footprint, deploying NFs requires the collaboration and coordination among multiple providers.

- **Privacy:** NFPs are known to be secretive about their networks. Information such as link utilization, servers' load and network topology is deemed confidential. For instance, Amazon EC2 [10], a widely-known cloud datacenter provider, does not disclose any information about servers' utilization or geographical location to clients but rather advertises a set of types where each type has a combination of physical resources (e.g., CPU, memory and network) and a service cost. Furthermore, in terms of topology information, ISPs typically reveal simplified graphs which neither show the router-level connectivity nor identify the PoPs structure of an ISP network [124]. Hence, NFPs should be able to collaborate for the deployment of network service while preserving the privacy of each provider.

- **Autonomy:** NFPs are usually administratively-independent domains which implement different policies to manage their network resources. For example, some NFPs might aim at achieving load balancing to provide better reliability and robustness, while others might aim at maximizing resource utilization and revenue. Subsequently, NFPs should be able to deploy NFs according to their internal policies, i.e., network service deployment should not enforce a single policy across the providers or violate the policies of the individual providers.

- **Middlebox traversal:** Despite the essential functionality they bring to the network, middleboxes introduce various undesirable implications on traffic that traverses them and hinders the establishment of connections when certain protocols are in use (e.g., Stream Control Transmission Protocol (SCTP) [125], Multipath TCP [131], UDP). For example, Network Address Translators (NATs) rewrite IP addresses and ports, proxies break end-to-end semantics, and firewalls may block UDP traffic or cache out-of-order-packets introducing varying delays. To mitigate these problems, most applications resort to tunnelling traffic over the

non-blocked protocols (e.g., HTTP, TCP or UDP) or to encrypting packets leading to high power consumption and waste of bandwidth. Most of these implications stem from the middleboxes deployed by access ISPs and cellular networks. Hence, to avoid tunnelling or encryption, the deployment of middleboxes should foster the collaboration between end-hosts and ISPs (or NFPs) such that end-hosts express their requirements for establishing connections and ISPs may redirect the traffic through a set of middleboxes fulfilling these requirements.

These challenges, in turn, raise a set of requirements for network service deployment:

- **Middlebox discovery:** For on-path network functions deployment, a prerequisite is the discovery of the middleboxes (i.e., processing platforms) and NFPs along the traffic path. Existing techniques for path discovery (e.g., *traceroute*, Tracebox [59]) incur delays that will substantially prolong flow processing establishment. Protocols for middlebox signaling have been developed primarily for device configuration (e.g., SIMCO [126] and NSIS [87]). Furthermore, AS-path retrieval from BGP routers will not augment middlebox discovery, since the sequence of middleboxes traversed by the flow within each NFP will still be required. Hence, to establish on-path processing, we need to design signaling protocols for middlebox discovery which incur low delay and maintain the privacy of each provider.

- **Network functions assignment:** We need to develop privacy-preserving approaches for assigning network functions to processing platforms (deployed on-path or on datacenters) and network links across multiple providers. In addition to privacy, these approaches should enable each provider to enforce its own policy (i.e., preserve autonomy) while maintaining the correctness of the service chain within and across NFPs. More specifically, this requires the design of algorithms for middleboxes selection and NFPs assignment (for on-path processing) as well as algorithms for network path selection and mapping NFs to servers and links within datacenters (for off-path processing).

- **Service chain partitioning:** As with any service offering, clients expect market competition and cost minimization for networks' service offering. In particular, depending on NFs location requirement, each provider's policy and available resources, NFPs should be able to compete for different sets of NFs (i.e., subchain) within a service chain and subsequently offer different service prices to the client. This raises the need for developing a service chain partitioning approach which enables competition

while preserving providers' privacy and autonomy as well as service chain correctness within and across the providers.

- **Invariant-preserving middlebox traversal:** To foster collaboration between end-hosts and the ISPs for the establishment of connections through middleboxes, end-hosts should be able to express their requirements as an invariant (e.g., using the API in [113]). For example, an end-host can request not modifying certain packet fields or payload, permitting UDP traffic or enabling access to public DNS servers. Upon the submission of such a request, the ISP may be willing to redirect the traffic through a set of middleboxes (e.g., NAT and firewall) that comply with its security policy and, at the same time, preserve the invariant expressed by the end-host. This, in turn, raises several requirements: (i) the collection of middlebox configurations, (ii) the selection of invariant-preserving middleboxes and shortest paths, and (iii) the installation of forwarding entries in the ISP's routers to route the traffic through the assigned path.

- **Workload profiling:** To consolidate NFs on commodity servers and enable admission control on service requests, the knowledge of the computational requirements of NFs' workloads is needed. Profiling packet processing workloads on commodity servers is challenging due to the use of optimization techniques such as polling and batch processing for maximizing processing rate. In more detail, polling enforces checks for new incoming packets, as frequently as possible, irrespective of the packet arrival rate leading to high CPU utilization, regardless of the traffic rate and the processing workload. On the other hand, batch processing, which is performed to reduce the computational requirements of a workload, becomes less effective for low traffic rates, since the number of packets per batch is smaller, resulting in high computation requirements per packet (i.e., cycles/-packet).

1.2 THESIS CONTRIBUTIONS

In this thesis, we provide the following contributions:

- To establish on-path network processing, we develop MIDAS, an architecture for the coordination of on-path processing setup which circumvents the difficulty in middlebox discovery and selection across multiple NFPs, without any prior knowledge of the network path. We propose signaling protocols for service middleboxes discovery and NFP interoperability. Furthermore, we design an order-preserving middlebox selection approach which maintains providers' privacy by using Multi-Party Computation

(MPC) protocol to preserve the confidentiality of middleboxes' utilizations across NFPs.

- We develop a prototype implementation to show the feasibility of our on-path flow processing architecture in terms of flow processing setup delay with a diverse range of middleboxes, NFPs, and network service request arrival rates. Our prototype comprises a Click Modular Router implementation of our middlebox discovery protocol, an MPC protocol for the middlebox selection approach and a software middlebox platform for consolidating NFs on a commodity server.

- We propose DistNSE, a distributed architecture which enables the collaboration between NFPs for off-path Network Service Embedding (NSE) while maintaining the privacy and the autonomy of the participating NFPs. DistNSE further ensures competitive pricing by enabling different providers to compete for different NFs of a service chain. Our architecture decomposes processing establishment into two steps: inter-provider service embedding and intra-provider service embedding. Inter-provider embedding provides a distributed algorithm for service chain partitioning and competition establishment. We couple our inter-provider algorithm with a communication protocol to exchange embedding information among NFPs. On the other hand, intra-provider embedding maps service chains to DCs and network links of an NFP while implementing two different policies: service cost minimization and load balancing.

- We present a Software-Defined Networking (SDN) architecture for invariant preserving middlebox traversal. Following the trend for (logically) centralized control, we rely on a centralized controller deployed by the ISP which retrieves middlebox configurations, selects middleboxes and paths that preserve the specified invariant, and sets up packet forwarding along the selected path. Middlebox checking against invariants can be performed using recent advances on static analysis, such as Header Space Analysis (HSA) [96] or SymNet [128]. For the installation of flow entries in routers, we employ OpenFlow [107]. Our work is mainly focused on middlebox and path selection. To this end, we present and evaluate an algorithm for the selection of a path through a set of invariant-preserving middleboxes.

- We investigate the implications and challenges of workload profiling on commodity servers, and we exemplify a technique for gauging the computational requirements of packet processing workloads. We apply instrumentation and develop a closed-loop control approach to measure workload computational requirements at a given processing rate, while minimizing the ef-

fect of workload-unrelated processing operations (e.g., empty polls) and maximizing the effect of workload-related operations (e.g., batch processing). Furthermore, we present the computational requirements of selected packet processing workloads and show the effect of various I/O optimizations on workload CPU utilization.

1.3 THESIS OUTLINE

The remainder of this thesis is organized as follows:

- **Chapter 2** presents the concepts and technologies that are relevant to understand this thesis.
- **Chapter 3** discusses the challenges of workload profiling on commodity servers and introduces our technique for gauging the computational requirements of packet processing workloads.
- **Chapter 4** reviews the implications of middleboxes on traffic and connection establishment and presents our SDN architecture for invariant preserving middlebox traversal.
- **Chapter 5** discusses the challenges and requirements of on-path processing and presents the different components of MIDAS, our architecture for on-path flow processing. This chapter further introduces our MIDAS prototype implementation.
- **Chapter 6** investigates the challenges and requirements of off-path NSE, and introduces the different components of DistNSE, our architecture for distributed off-path network service embedding.
- **Chapter 7** highlights the conclusions of this thesis and presents our outlook on future work.

2 BACKGROUND

In this chapter, we introduce the concepts and the technologies that are essential to understand this thesis. We start by defining the role middleboxes play in today's network. Next, we present the concept of service chaining as a new approach to represent the deployment policy of middleboxes. Then we give an overview of Network Function Virtualization (NFV) concept. Subsequently, we review the different technologies facilitating the realization of NFV. In particular, we discuss the advancement of commodity servers' capabilities and particularly focus on packet I/O handling optimizations. We further present different software packages and virtualization techniques that can be used to implement and consolidate NFs on commodity servers. In the last section of this chapter, we discuss Software-Defined Networking (SDN), a technology which complements NFV to provide steering and routing through NF service chains.

2.1 MIDDLEBOXES

Middleboxes have become more popular than ever providing various functions in enterprise, cellular and ISP networks. A recent study found that the number of middleboxes in enterprise networks is as high as the number of routers and switches [122]. Wang *et al.* [130] reported that 72 out of the 107 cellular networks they studied rely heavily on NATs to scale beyond the limited IPv4 address space. Furthermore, recent years have seen a significant growth in the middlebox market. For instance, the global firewall market has increased to $9.5 billion in 2014, showing 9.5% increase over 2013. The market for application delivery network (e.g., WAN optimizers, loadbalancers, application gateways) will exceed $6.2 billion by 2020 [7].

According to RFC3234 [51], middleboxes are defined as network appliances interposed on the path between a source node and a destination node to provide network functions other than forwarding and routing. Middleboxes are very diverse in terms of function and application, performing a wide range of tasks (e.g., filtering, inspection, transformation, redundancy elimination, caching and encryption) and processing packets at different networking layers from network to application layer. Due to this diversity, forming a comprehensive classification of middleboxes is deemed challenging. RFC3234 [51] provides an approximate taxonomy for middleboxes by defining a set of facets with which a middlebox is classified. Examples of these facets include the protocol layer at which the middlebox pro-

cesses packets, the visibility to the end nodes, the purpose of the middlebox as a functional or an optimization network device, and whether a middlebox function is processing or routing. The assignment of facets to middleboxes is to some extent subjective and might vary depending on the point of view. For example, some might consider a NAT a functional middlebox, while others might see it as an optimization device to overcome the depletion of IPv4 address space. Nonetheless, to illustrate their role in the network, we present some common examples of middleboxes:

- **Firewall:** A firewall filters packets based on a predefined set of rules configured by the network administrator. In principle, a firewall rule defines a set of header fields (e.g., IP source/destination address, source/destination port number, protocol) and an action (e.g., allow, deny). Based on the header fields, the firewall identifies the packets on which the action is performed. A firewall can be stateful or stateless. A stateless firewall performs filtering without maintaining any state about the arriving packets. On the other hand, a stateful firewall runs with a complex set of rules which classifies packets into connections (e.g., TCP flow) and maintains the state per connection such that, in addition to malicious packets, illegitimate connections are also blocked.

- **Network Address Translator (NAT):** A NAT enables the sharing of a public IP address among multiple end nodes with private IP addresses by mapping the private IP address and the source port number of an end node connection (TCP or UDP) to the public IP address and a selected port number [66]. The main function of a NAT is to mitigate the problem of IPv4 address space depletion. As discussed earlier, NAT is one of the most popular middleboxes in cellular and ISP networks [130].

- **Intrusion Detection System (IDS):** An IDS monitors and analyses the network traffic to identify malicious activities and violations of the network security and policies and subsequently takes an action (e.g., alert the network administrator). Intrusion detection approaches are categorized into three main groups [104]: (i) signature-based detection, which identifies intrusion by comparing the observed events against a database of known attacks and threats, (ii) anomaly-based detection, which detects intrusion by identifying events that deviates the system from its expected or normal behaviour, and (iii) stateful protocol analysis, which keeps track of a protocol's state (e.g., TCP) to recognize events that don't match a protocol's predefined behaviour. Snort [31] and Bro [11] are two popular open source IDS tools based on signature-based and anomaly-based methodologies.

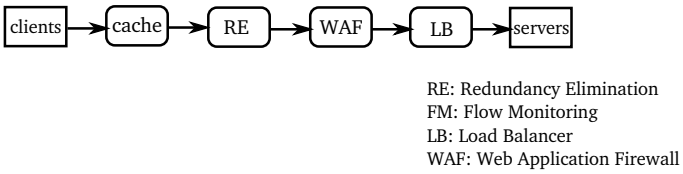


Figure 1: Example of a service chain.

- **Proxy:** A proxy is an intermediary device that processes requests sent to servers by clients to provide various functions such as caching [56, 48, 33], compression, content logging and monitoring, content-based filtering, load balancing, flow aggregation and splitting [52]. In principle, proxies operate at and between the transport and application layer aiming at improving communication performance (e.g., minimize access delay, conserve access bandwidth), providing content access control (i.e., filter unauthorised access to certain web pages or content), optimizing resource utilization (e.g., distribute HTTP requests among different web servers, offload SSL transactions [94]), and/or protecting servers or clients against security attacks.

2.2 NETWORK SERVICE CHAINING

Middlebox deployment is commonly driven by certain policies, i.e., traffic should traverse a set of middleboxes in a specific order. For example, traffic should traverse a firewall before reaching a cache to prevent unauthorised access. Traditionally, the order of middleboxes is enforced by physically placing them at particular points on the network. With the emergence of NFV where middleboxes are software modules which can be migrated, duplicated or removed depending on network load and structure changes, a flexible approach is needed to represent the order of middleboxes. To this end, the concept of service chaining has become popular [111][72] [112]. A service chain is a directed graph with edge nodes representing the traffic source and destination, intermediate nodes representing the NFs and links showing the order of middleboxes. Figure 1 gives an example of a service chain where clients access a set of web servers. In this example, to minimize delay, the client traffic access the cache first followed by the redundancy elimination which conserves the bandwidth required to communicate with the servers. Furthermore, before reaching the load balancer, the traffic must be checked by the Web Application Firewall (WAF) to filter malicious web access.

2.3 NETWORK FUNCTION VIRTUALIZATION

Network Function Virtualization (NFV) is a new concept aiming at replacing the special purpose hardware-based middleboxes with software-based NFs running on virtualized infrastructures [13] [34] [35]. More specifically, NFV enables the consolidation of software NFs on standard commodity servers, [64] switches and storage which could be placed in the ISP network nodes, datacenters and end users properties. This further involves migrating, instantiating and cloning NFs in different places in the network as needed without requiring the deployment of new hardware.

As a result, NFV brings a high level of flexibility and programmability to the network leading to a shorter time-to-market through software-based development, the involvement of more players (e.g., small businesses and academia) in NFs innovation, and customizable network services that can be tailored to clients' geographical locations and preferences. Furthermore, as with cloud computing, outsourcing NFs to datacenters opens the door for new service models such as NFaaS [13]. NFaaS enables network operators to rent their compute resources to host clients' NFs in a pay-per-use fashion leading to operation and investment cost saving and dynamic on-demand-based resource provisioning.

To implement NFV, ESTI NFV defines three architectural components [13]:

- Virtualized Network Function, which is the software implementation of a network function that can be hosted on the NFVI.
- NFV Infrastructure (NFVI), which is comprised of the different physical resources that should be virtualized to run the NFs.
- NFV Management and Orchestration, which orchestrates and manages the lifecycle of the physical and/or the software resources of the NFVI and the NFs. NFV Management and Orchestration performs tasks such as network service embedding, resources provisioning, configuration, and monitoring.

The realization of these three components (i.e., NFV) is attainable through the evolution of several recent technologies such as the advancement on commodity servers capabilities providing high performance processing platforms, the optimization of packet I/O engine aiming to accelerate packet handling, the development of software frameworks and packages to design and implement NFs, and the availability of diverse virtualization techniques to consolidate NFs. Furthermore, NFV can benefit from SDN techniques and concepts. In particular, SDN can provide traffic redirection and steering through NFs to comply with the service chains' specifications. In the following sections, we will discuss the NFV enabling technologies in more detail.

2.4 PACKET PROCESSING ON COMMODITY SERVERS

Traditionally, middleboxes were developed to achieve high performance and perform single-purpose limited packet processing functions by relying on specialized and closed software and hardware (e.g., custom ASICs, network processors). Extending or programming such equipment requires hardware redesign and upgrade as well as special programming skills (e.g., programming network processors [68]). However, with the emergence of sophisticated and frequently evolving network functions, there has been a growing need for extensible and programmable middleboxes. As a result, a significant amount of work has been dedicated to build packet processing platforms using commodity servers [119, 64, 86, 40, 65, 115, 133, 67, 99, 106].

Commodity servers provide an unprecedented level of programmability and flexibility [119, 64, 81]. They are based on hardware, operating systems and software with which a large base of developers are familiar. This, in turn, facilitates changing or extending network functions through software-only upgrades, obviating the need for hardware design and development. Furthermore, commodity multicore CPUs with instruction sets optimized for certain NFs and GPUs with massive level of parallelism [86] offer tremendous computational power and a high level of parallel processing which enables a multitude of packet processing operations and high forwarding rates. This processing power is supported by large caches and a dedicated memory controller offering high memory bandwidth and low data access delay. To receive and send packets, commodity servers are equipped with high speed Network Interface Cards (NICs) that provide a capacity of up to 40 Gbps per port/link and a high I/O bandwidth bus (provided through Peripheral Component Interconnect express (PCIe)) to transfer packets between NICs and the memory. This is further complemented by multi-queueing technologies providing line speed packet classification in hardware.

These capabilities of commodity servers have been further augmented with remarkable advances on packet I/O handling. Technologies such as Netmap [115], DPDK [1] and Routebricks [64] significantly reduce the computational overhead of packet I/O handling on commodity servers by employing techniques such as pooling, batch processing, zero copying and minimum data structuring. This, in turn, leads to a higher packet forwarding rate and efficient resource utilization. We will discuss these technologies in more detail in section 2.4.2.

Moreover, to facilitate the implementation of software NFs on commodity servers, several packet processing software packages have been developed (e.g., Click Modular Router [99], Snort [31]). These packages enable the design and implementation of various packet

processing operations such as encryption, intrusion detection and prevention, packet aggregation, and redundancy elimination. Section 2.5 provides further details about different NF software packages

2.4.1 *Packet I/O on Commodity Servers*

Packets I/O handling is one of the essential operations of any packet processing workload. It further represents a significant portion of the workload's computational requirements. Therefore, in this section we provide an overview of the basic operations of packet I/O handling.

As shown in Figure 2, Packet I/O is basically comprised of the receive and the transmit path.

Receive Path. The receive path essentially consists of two operations: (i) transferring packets from the NIC to the main memory using Direct Memory Access (DMA) (arrow 1 in Figure 2) and (ii) accessing the packets copied to the memory by the CPU (arrow 2 in Figure 2). In more detail, the DMA transfers the packets from the NIC hardware queues to a chained ring of buffers, called Rx ring, allocated in the main memory. Each Rx buffer is managed through a data structure called receive (Rx) descriptor. The Rx descriptor stores information about the packet data buffer address, length, checksum and status. The completion of a packet transfer to the Rx ring is announced through the status field. As soon as the packets are copied into memory, the CPU accesses and processes each packet.

Transmit Path. Akin to the receive path, the transmit path is comprised of two operations: (i) upon processing, the CPU writes packets to memory (arrow 3 in Figure 2) and (ii) copies packets from the main memory to the NIC using DMA (arrow 4 in Figure 2). In particular, the CPU places the processed packets in another chained ring of buffers, called Tx ring, allocated in the main memory. Then, the packets are transferred from the Tx ring to the NIC hardware queue. The Tx ring buffers are managed through a data structure, called the transmit (Tx) descriptor, which holds the packet data buffer address, length, checksum and status. Similar to the Rx descriptor, the completion of a packet transmission (i.e., transferred to the NIC) is announced through the Tx status field. Upon packets transmission, the Tx buffers are recycled and reused for the transmission of subsequent packets.

A crucial aspect of packet I/O handling is the interaction between the NIC and the CPU. In particular, the CPU should be able to know when new packets are copied into memory and when their transmission has been completed. Traditionally, the arrival or departure of packets has been announced by issuing interrupts to the CPU. However, since interrupts yield low packet forwarding performance, mod-

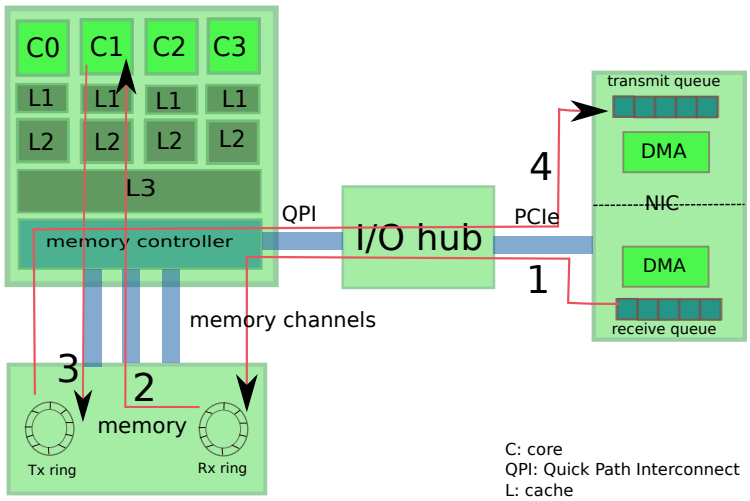


Figure 2: Packet I/O handling on commodity servers.

ern packet processing systems typically rely on polling. Polling eliminates the processing overhead incurred by interrupt handling and prevents receive livelock.

2.4.2 *Advanced Packet I/O Engines*

In the recent years, several works have developed techniques to speed up packet I/O handling and maximize forwarding rates [115, 64, 1, 58, 74, 5]. In this section, we review the optimization techniques presented in three works:

- **Routebricks:** Routebricks [64] relies on polling and batch processing to optimize the packet I/O handling rate. In particular, packets are being polled in batches from the Rx/Tx ring eliminating the interrupt handling cost and leading to lower per-packet bookkeeping overhead (i.e., overhead to update Tx/Rx descriptors). Furthermore, to ensure the availability of a sufficient number of packets in the Rx/Tx rings, Routebricks also allows moving packets from the NIC queues to the rings in batches resulting in fewer PCIe and I/O transactions. In addition to polling and batch processing, Routebricks distributes each NIC traffic across different CPU cores by assigning a separate NIC hardware queue to each CPU core. This results in efficient utilization of the CPU computational capacity and avoids the expensive operations of locking and unlocking when a packet is accessed by different cores (i.e., a packet is always processed by a single core).

- **Netmap:** Netmap [115] achieves high performance packet I/O handling through i) lightweight descriptors that enable the processing of a large number of packets in a single system call re-

sulting in bookkeeping cost amortization, ii) static linear packets' buffers allocation leading to buffer allocation and deallocation overhead saving iii) eliminating data-copy overhead by giving application protected and direct access to packet buffers, and iv) polling and batch processing for accessing and transferring packets.

- **Intel DPDK:** Similar to Netmap and Routebricks, DPDK [1] optimizes packet handling through polling, batch processing and static buffer allocation. Furthermore, it provides a queue manager to implement lockless queues which enables different cores to access packet data without the need for performing expensive lock operations.

2.5 SOFTWARE PACKAGES FOR NETWORK FUNCTIONS IMPLEMENTATION

As a result of the advancement on commodity servers performance and capabilities, many software packages have been developed to implement and run software network functions. Examples of such packages include snort [31] and bro [11], which implement intrusion detection and prevention functions; Squid [33], a web caching proxy to conserve bandwidth and minimize access delay; IPtables [16], which supports filtering and transformation functions such as firewall and NAT; and Click Modular Router [99], a modular framework that facilitates the implementation of various types of functions (e.g., NAT, firewall, encryption). In this thesis, we mainly rely on Click to implement different types of NFs. In the following section, we will present an overview of Click and its packet I/O handling techniques.

2.5.1 *Click Modular Router*

Click [99] is a modular software package widely used for the implementation of packet processing systems on commodity servers. Click offers a wide range of packet processing elements (e.g., packet I/O, queueing, table lookup), which can be connected in a graph, allowing the construction of a customised data plane. The Click elements that compose a given data-plane configuration can be assigned to multiple threads, allowing a high level of parallelism with multi-core CPUs. Click can run in the Linux kernel offering high packet forwarding rates.

With respect to packet I/O, Click relies on two elements: *PollDevice* for reception and *ToDevice* for transmission. Whenever *PollDevice* is scheduled, it issues a NIC device driver call which checks the status field of the Rx descriptors for new received packets. Subsequently, the driver call passes the new packets along with their metadata struc-

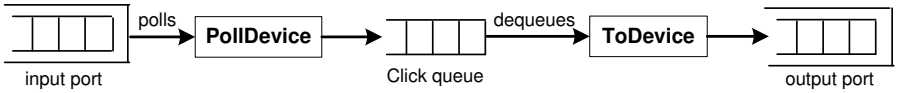


Figure 3: Exemplary Click forwarding path.

tures (*sk_buff* in Linux) from the Rx ring to *PollDevice*. *PollDevice* can be configured to poll packets in batches (i.e., batch processing).

On the transmit path, *ToDevice* passes packets enqueued by Click to the Tx ring. Akin to *PollDevice*, packets may be transferred to the Tx ring in batches. *ToDevice* receives information from the NIC device driver regarding the status of packet transmission and upon its completion, *ToDevice* recycles the Tx ring buffers. Finally, the NIC device driver returns the recycled buffers to the Rx ring.

An exemplary Click forwarding path comprising a *PollDevice*, a queue, and a *ToDevice* is illustrated in Figure 3. *Polls* represent the packet transfers from the Rx ring by *PollDevice*, while *dequeues* denote the packets fetched from the Click queue by *ToDevice*.

2.6 SERVER VIRTUALIZATION TECHNOLOGIES

Virtualization has become an essential technology for the IT industry. Today, more than 50% of servers are virtualized and this number is expected to increase to 86% by 2016 [36]. The popularity of virtualization comes from its numerous advantages. In particular, virtualization enables the consolidation of many workloads on a single physical machine leading to saving in power, space and computing resources. Furthermore, virtualization provides dynamic load balancing and failure recovery by facilitating the migration of a virtual machine from an overloaded or failed physical machine to machines with low load and healthy hardware. By isolating different workloads and users in separate virtual machines, virtualization also creates a reliable and secure environment for developing and testing new applications. Additionally, virtualization is the key technology for cloud computing which allows enterprise networks to achieve significant savings on operation and investment cost by migrating their applications to virtualized machines deployed in external datacenters.

In principle, virtualization is classified into process and system virtualization [123]. Process virtualization creates a virtual machine by starting and executing an operating system process. Typically, the goal of such a virtualization is to provide a platform-independent programming environment to execute programs across different platforms. On the other hand, system virtualization provides a full system environment for executing a complete operating system and its user processes [123]. Through System virtualization, multiple Virtual Machines (VMs) running different or the same operating systems can coexist on a single machine.

System virtualization is realized through a set of approaches:

- **Full virtualization:** Full virtualization creates a virtual machine which fully emulates the underlying hardware to the guest operating system [38]. Subsequently, the guest operating system does not have to be aware of the virtualization, and it requires no modification. This means an OS can be migrated and ported across different physical machines without the need for re-compilation or adaptation. Common examples of full virtualization technologies include VirtualBox [37], VMware Workstation [39] and QEMU [26].

- **Paravirtualization:** Due to the overhead incurred by full virtualization to emulate the physical hardware, paravirtualization is another system virtualization approach where the guest operating system is modified to fit the virtualized environment. Paravirtualization allows non-virtualized access to phyiscal resources obviating the need for interception and translation usually required for a virtual access. As a result, paravirtualization achieves higher performance and incurs less overhead than full virtualization, however, significant modification needs to be applied to the guest operating system. A common example of paravirtualization is XEN [42].

- **OS-level virtualization:** OS-level virtualization aims at providing high performance and low virtualization overhead by partitioning machine hardware resources at the operating system level. In particular, this approach creates a set of lightweight virtual containers, with an isolated set of resources (memory, CPU, disk) and processes, which share the same kernel of the host. This, in turn, restricts containers to only run an operating system that is compatible with the host OS, i.e., a host running Linux OS can only run Linux-based guest OSes on its containers. Examples of OS-level virtualization software are LXC [17], OpenVZ [24] and Docker [32].

2.7 SOFTWARE DEFINED NETWORKING

Software-Defined Networking (SDN) is a new networking architecture which decouples the network control plane from the data plane, moves network state and intelligence to a programmable (logically) centralized controller, and provides an abstract view of the network infrastructure to the applications [29, 30]. SDN delivers a set of compelling benefits to the network. In particular, SDN enables network operators and administrators to program the network control logic in real time leading to a shorter innovation cycle as well as a faster and more dynamic response to users' requirements and business needs.

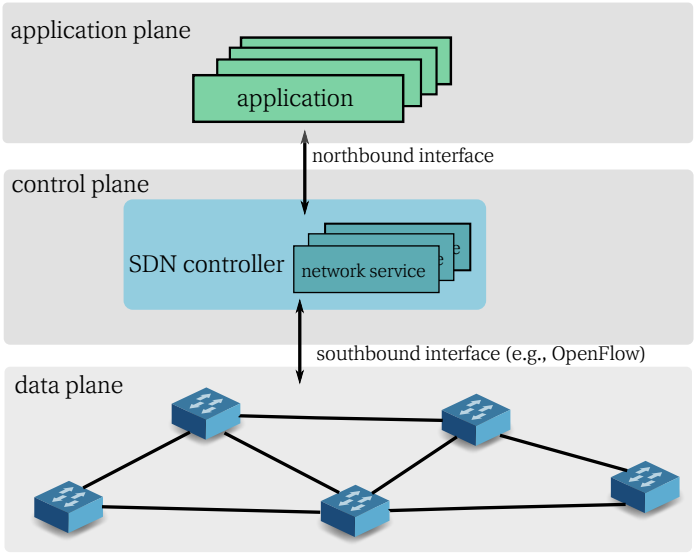


Figure 4: High-level overview of SDN architecture.

The SDN centralized controller further maintains network-wide visibility that abstract the network as a single logic switch to the applications and services. Configuring the network is much simpler with SDN because SDN provides a standard vendor-independent interface to access network devices obviating the need for a long complex configuration targeting diverse types of devices and vendors and leading to more consistent policy enforcement.

As Figure 4 shows, the SDN architecture consists of four main layers:

1. **Data plane:** The data plane is comprised of a set of simple switches. In comparison to the standard network devices, these switches do not run complex algorithms or calculations (e.g., routing algorithms) or implement sophisticated protocols, but they are rather dumb switches which perform packet forwarding based on the instructions sent by the control plane.

2. **Control plane:** The control plane is the core component of the SDN. As discussed above, the control plane provides a platform (i.e., SDN controller) to run the software modules which implement the network control logic (e.g., routing, loadbalancing, access control). In Section 2.7.1, we will discuss several existing SDN controllers.

3. **Application plane:** The application plane represents the business or user applications in need for network services support. For example, an application could be a video application requesting routing through a network path with low delay.

4. **Southbound interface:** The southbound interface facilitates communication between the switches and the controller. It basically

enables configuring forwarding rules in the switches, collecting statistics about the traffic characteristics (e.g., BW) and switch operation, advertising switches capabilities and announcing new events (e.g., arrival of a new packet). The most popular example for a southbound interface is OpenFlow. We will provide an overview of OpenFlow in Section 2.7.2.

5. **Northbound interface:** The northbound interface enables communication between SDN applications and the SDN Controller. In principle, the northbound interface creates an abstractive view of the network such that applications can express their networking requirements in a simple and direct way.

2.7.1 *SDN Controllers*

The recent years have seen the development of many SDN controllers with various features and capabilities [101, 108]. Kreutz *et al.*, in their SDN survey [101], report more than 25 different controllers. When comparing different SDN controllers, many aspects need to be considered. These aspects include the controller architecural design approaches (e.g., centralized or distributed), north and southbound interface technologies, the controller's core functions, performance, and programming language [101]. For instance, while controllers such as POX [25], Ryu [28], Beacon [71] and Floodlight [14] implement the control logic as a single unit, other controllers such as Onix [100], OpenDaylight [23] and ONOS [21] enable the distribution of control logic across multiple controller instances (deployed on different machines or locations). Distributed controllers provide more resilience and robustness against failures and faults while enabling scale-up to handle high network load. However, distributed controllers also require state synchronization across instances to maintain network consistency. So far, distributed controllers (excluding Onix, ONOS and SMaRtLight [46]) do not enforce strong consistency between instances [101], which, in turn, might negatively impact network applications with high consistency requirements (e.g., load balancing [103]).

The majority of existing controllers exclusively implement OpenFlow as a southbound interface. Yet, few controllers (e.g., OpenDaylight, Onix) also support other southbound interfaces such as NetConf [69], BGP, SNMP and OVSDB [109], allowing backward compatibility with regular routers and switches and enabling access to other network devices (e.g., firewalls and NATs). In terms of the northbound interface, there is a greater diversity among controllers, i.e., as opposed to OpenFlow being a widely adopted southbound interface, there has been no common northbound interface [101, 108]. In fact, the majority of controllers (e.g., OpenDaylight, Onix and Floodlight) design and specify their own northbound interfaces. A signifi-

cant number of these interfaces are based on the REST architectural style (e.g., OpenDaylight, ONOS, Floodlight).

Similar to operating systems which offer services such as I/O access, memory management, scheduling and isolation to application programs; controllers need to support network applications with functions such as topology discovery, statistics collection, shortest path forwarding and switch management. Many existing controllers (e.g., OpenDaylight, Floodlight and POX) offer a broad range of general-purpose functions to network applications. Other controllers further introduce functions to support special purposes, e.g., Onix provides consistency and state management to enable distributed control, and SoftRAN [82] offers radio spectrum allocation to allow LTE access network management. However, as more network applications and SDN use cases emerge, controllers are expected to offer more services and functions.

Whether it is a datacenter, an enterprise or an ISP network, performance is a key factor when selecting a controller for managing a network. A controller's performance is typically evaluated based on the controller's throughput (i.e., the number of processed flows per second) and latency (i.e., response time to switches requests). While there is no comprehensive study on the performance of existing controllers, some studies report the performance of a small set of controllers. For instance, a recent study [120] evaluated the performance of six controllers: POX, Floodlight, Beacon, MuL [18], Maestro [49] and Ryu. This study found that Beacon achieves the maximum throughput with 7 million flows per second, whereas MuL incurs the lowest latency. Another study [71] also showed that Beacon outperforms Maestro, Ryu, Floodlight and POX with a throughout of 12.8 million flows per second using 12 CPU cores. The performance of a controller relies on various aspects such as its programming language, being single-threaded (e.g., POX) or multi-threaded (e.g., Beacon, NOX and Floodlight), the multi-threading scheduling mechanism (e.g., Beacon uses round-robin scheduling, NOX statically pins switches to threads [120]) and the implementation of the basic functions and services (e.g., event handling, lookup table). However, there is a trade-off between a controller's performance and other aspects such as being developer-friendly. For instance, POX, which achieves low performance in comparison to other controllers [120], has been popular in the research community for rapid network services prototyping [111, 77, 76, 91], due to its easy-to-use and Python-based framework.

2.7.2 *OpenFlow Protocol*

OpenFlow [107] provides an open API to control the forwarding table in switches and routers. OpenFlow-enabled switches maintain a flow

table which is configured by a remote controller over a secure connection using a set of OpenFlow commands. Each flow table entry consists of a match rule which defines a flow based on a set of matching packet header fields (Table 1), a corresponding action which specifies a set of packet forwarding and processing operations, and statistics which monitor the number of arriving packets and bytes as well as the time passed since the last packet matched the flow rule.

OpenFlow defines three basic actions for processing packets: i) *forward packet* to a switch port or ports which constitutes the basic operation to route packets through the network, ii) *drop packet* which can be used to filter untrusted packets, limit broadcast domains, or support protocols operations such as dropping IP packets with expired TTL, and iii) *encapsulate and send a flow packets* (or the first packet) to the controller where a decision about the flow can be made (i.e., add a flow entry for forwarding or dropping the flow's packets). Furthermore, OpenFlow supports matching packets across a wide range of packet header fields and protocols (e.g., IP, TCP, UDP, VLAN, Ethernet, MPLS). A matching header field can designate a specific value or be a wildcard facilitating flows aggregation, e.g., all flows carrying a particular destination IP address are matched as one single flow.

One of the popular switches implementation that supports OpenFlow is OpenvSwitch [110]. OpenvSwitch is an open source switch which provides multilayer processing and exposes an external interface for programmability, control and configuration. It is typically deployed within the hypervisor to forward and steer packets across virtual machines and the physical interfaces. In addition to OpenFlow, OpenvSwitch supports standard management interfaces and protocols such as NetFlow, sFlow, RSPAN and CLI.

Input port	MAC src	MAC dst	Eth type	VLAN ID	IP src	IP dst	TCP sport	TCP dport

Table 1: OpenFlow matching fields.

3

PACKET PROCESSING WORKLOAD PROFILING

In the recent years there has been a growing interest in building software middleboxes using general-purpose commodity servers. Commodity servers offer ample processing power which in conjunction with the availability of packet processing software (e.g., Click Modular Router [99], Snort [31]) enable performing various packet processing operations beyond IPv4 forwarding such as encryption, intrusion detection and prevention, packet aggregation, and redundancy elimination.

However, in order to improve the resource utilization of commodity servers, provide high level of consolidation and enable admission control on incoming processing requests, the knowledge of packet processing workloads' computational requirements is needed. Former studies [64, 86] have presented the computational requirements of selected packet processing workloads such as IPv4 forwarding and IPsec. Nevertheless, workload computational requirements may change depending on packet I/O techniques and optimizations (e.g., packet handling in batches) [64, 86, 115]. As a result, previously reported workload computational requirements may not be applicable to packet processing systems where a different I/O technique is exercised.

Profiling packet processing workloads on commodity servers is not trivial. For example, high-performance packet processing systems typically perform polling to handle arriving packets. This entails significant implications on workload profiling, since the CPU appears to be fully utilized, irrespective of the processing load. In this chapter, we discuss the implications and challenges of workload profiling on commodity servers, and we exemplify a technique for gauging the computational requirements of packet processing workloads. Furthermore, we present the computational requirements of selected packet processing workloads and show the effect of various I/O optimizations on workload CPU utilization. Our workload profiling technique is implemented on a packet processing platform based on Click (see Section 2.5.1). However, our profiling method is widely applicable to any packet processing system that uses polling and batch processing.

The remainder of this chapter is organized as follows. In Section 3.1, we discuss the challenges of workload profiling. We present our profiling technique in Section 3.2. In Section 3.3, we discuss our workload profiling results. Section 3.4 provides an overview of related work. Finally, we highlight our conclusions in Section 3.5.

3.1 PROFILING CHALLENGES

The difficulty in workload profiling mainly stems from the implications and side-effects of polling and batch processing, which comprise common techniques for packet I/O handling on commodity servers, as discussed in Section 2.4.1 and 2.5.1.

Polling. Polling enforces checks for new incoming packets, as frequently as possible, irrespective of the packet arrival rate. Therefore, polling utilizes all idle CPU cycles, regardless of the traffic rate and the packet processing application. Similar to polling, transferring packets to the Tx ring (i.e., *ToDevice* Click element, see Section 2.5.1) also tends to consume idle CPU cycles in order to speed up the packet transmission. Consequently, the full CPU utilization during packet transactions along the receive and transmit path inhibits the accurate estimation of workloads' computational requirements.

Batch Processing. Processing packets into batches reduces significantly the processing cost per packet. For example, multiple packets can be polled at once from the Rx ring, reducing the bookkeeping cost per packet. Batch processing can also be exercised at the application layer (e.g., encrypting packets in batches) or at the NIC to mitigate the PCIe transaction cost per packet. However, batch processing becomes less effective for low packet forwarding rates, since the number of packets per batch is smaller. Certainly, the effect of batch processing can be optimized by reducing the number of CPU cycles allocated to the workload. However, this may increase latency and result in packet drops. Essentially, the variable processing cost per packet, depending on the effectiveness of batch processing, complicates workload profiling.

It is important to note that these factors do not affect workload profiling in isolation. In fact, we need to take into account the confluence of polling and batch processing on workload profiling. A major challenge in this respect is to estimate the computational requirements of a workload under near-optimal operation, where the CPU resources entitled to the workload maximize the batch processing effectiveness without causing packet loss.

3.2 WORKLOAD PROFILING METHODS

Based on the observations in Section 3.1, we investigate methods to overcome the difficulty of profiling packet processing workloads on commodity servers. First, we define *empty polls* as the number of accesses to the Rx ring (initiated by *PollDevice*, see Section 2.5.1) in which no packet is found. Similarly, we define *empty dequeues* as the

number of attempts to fetch packets from the Click queue (initiated by *ToDevice*) where no packet is returned. A Click-based packet processing system typically yields *empty polls* and *empty dequeues*, due to the effect of polling.

Since polling results in full CPU utilization, there are two main approaches for the computation of workloads' requirements: (i) to estimate and deduct the number of CPU cycles consumed for all tasks that are not associated with the workload, and (ii) to restrict the cycles entitled to the workload such that *empty polls* and *empty dequeues* are minimal while the packet forwarding rate matches the target rate. Both approaches require the monitoring of *empty polls* and *empty dequeues*. To this end, we implement counters for both variables and the number of packets transferred within each batch, so that we can quantify the degree of batch processing.

According to the first approach, we merely have to estimate the number of cycles consumed by *empty polls* and *empty dequeues*, and subsequently deduct them from the nominal cycles of the CPU core on which the workload runs. In particular, this amount can be estimated by measuring the number of cycles utilized by an *empty poll* and *empty dequeue* and multiplying them by the number of monitored *empty polls* and *empty dequeues*. We found that this approach provides accurate results only for high packet forwarding rates (i.e., more that 10^6 packets/sec for IPv4), where batch processing is effective. When the polling rate is substantially higher than the receiving rate, the small number of packets per batch increases the computational cost per packet. Figure 5 illustrates the packets per batch versus the packet rate. It can be observed that for rates below 0.6 million packets/sec batch processing is eliminated. Note that this method was used by RouteBricks [64] (but without taking the *empty dequeues* into account) in order to compute the CPU cycles/packet for selected packet processing workloads at the maximum packet forwarding rate.

Due to the inefficiencies of this method, we rely on the second approach which allows us to obtain accurate results for a wide range of packet forwarding rates. Instead of estimating the number of cycles consumed by a workload, we aim to adjust the CPU resources entitled to the workload, such that the following conditions are satisfied: (i) there is no perceptible reduction in the packet forwarding rate, (ii) the number of *empty polls* and *empty dequeues* is minimal, and (iii) the number of packets per batch approximates an optimal value (i.e., 16 packets in our system).

We use an auxiliary workload to iteratively achieve the transition to this state. In particular, as auxiliary workload we use a Click forwarding path that comprises a *PollDevice* element. In principle, any workload that stresses the CPU and remains active for the whole measurement period can be used. The measurements are conducted on a single core which is shared by the packet processing and auxiliary

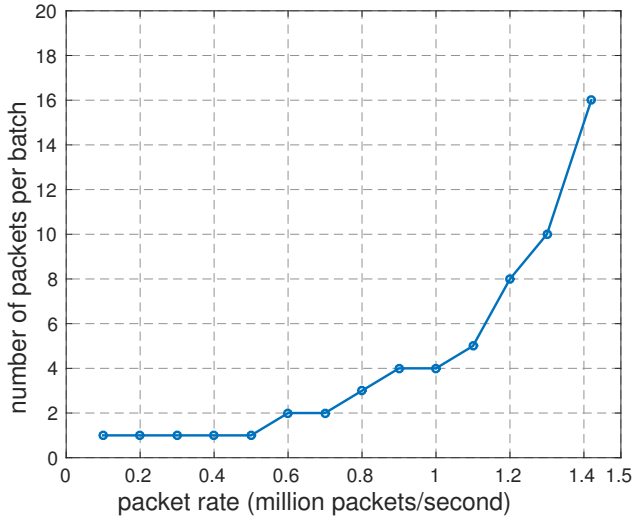


Figure 5: Average number of packets per batch vs. packet rate.

workloads. We iteratively adapt the resources entitled to the auxiliary workload by adjusting the scheduling priorities accordingly. During each iteration, we examine the packet rate, *empty polls*, *empty dequeues* and packets per batch to identify whether the system has approached the desired state. Once the packet processing system has reached this state, the number of CPU cycles consumed by the packet processing workload (W_p) corresponds to its minimum computational requirements, since the effect of polling on CPU utilization has been eliminated while batch processing has maximized its effectiveness.

Our workload profiling method is depicted in Algorithm 1. We initially seek to minimize the rate of *empty polls* (EP) and *empty dequeues* (ED) while achieving the target forwarding rate, and we subsequently record the current CPU utilization. Then we adjust the workload's CPU cycles in order to maximize the packets per batch (B). If we manage to optimize the effect of batching while maintaining a minimal number of EP and ED, we have reached the desired state and thereby measure the CPU cycles of W_p; otherwise, the algorithm returns the previously recorded sub-optimal CPU utilization. Since it is not feasible to achieve zero EP and ED as well as an optimal B, as seen from experiments in our packet processing platform, we use thresholds for all these values. In our system, we have set $\mathrm{EP}_{\mathtt{thresh}}$ and $\mathrm{ED}_{\mathtt{thresh}}$ to 40000 empty_polls/empty_dequeues per sec (which is a sufficiently low value compared to a maximum measured value of 3.5M), while $\mathrm{B}_{\mathtt{thresh}}$ is adjusted to 10 packets.

3.3 EXPERIMENTAL RESULTS

In this section, we present the computational requirements of selected packet processing workloads using our workload profiling technique.

Algorithm 1 Workload Profiling

```
while True do
    if W_p.Rate > TargetRate then
        decrease W_p.Cycles
        if (W_p.EP ⩽ EP_thresh) and (W_p.ED ⩽ ED_thresh) then
            MeasuredCycles = W_p.Cycles
            if W_p.B ⩾ B_thresh then
                return W_p.CPU
            else
                while W_p.Rate > TargetRate do
                    decrease W_p.Cycles
                    if W_p.B ⩾ B_thresh then
                        return W_p.Cycles
                return MeasuredCycles
    elseif W_p.Rate ⩽ TargetRate then
        Increase W_p.Cycles
        if (W_p.Rate > TargetRate) and (W_p.EP ⩽ EP_thresh) and
        (W_p.ED ⩽ ED_thresh) then
            MeasuredCycles = W_p.Cycles
            if W_p.B ⩾ B_thresh then
                return W_p.Cycles
            else
                while W_p.Rate > TargetRate do
                    decrease W_p.Cycles
                    if W_p.B ⩾ B_thresh then
                        return W_p.Cycles
                return MeasuredCycles
```

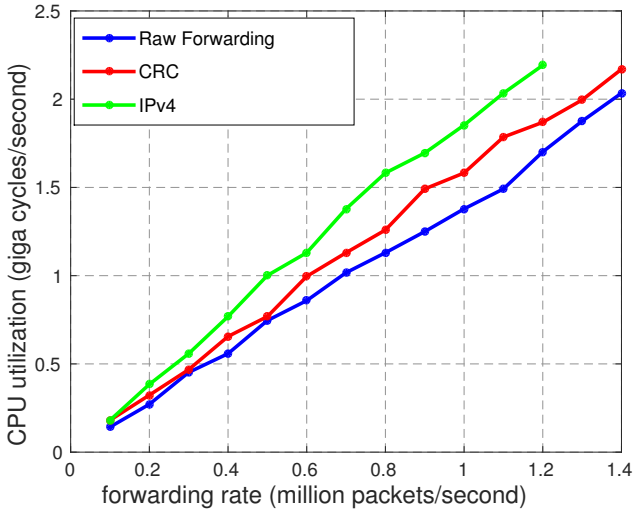


Figure 6: CPU cycles/sec for raw forwarding, IPv4 and CRC vs. packet forwarding rate with 64-byte packets.

Our experimental setup consists of three commodity NUMA-based (Non-Uniform Memory Access) servers (i.e., source, packet processing platform and sink), each equipped with a Xeon E5520 quad-core CPU at 2.26 GHz. Each core has a dedicated 32 KB L1 and 256 KB L2 cache, while all four cores share a 8 MB L3 cache. Furthermore, each server has 6 GB of DDR3 RAM at 1333 MHz and two quad 1G port NICs based on Intel 82571EB. Traffic is generated by a quad 1G port NetFPGA card. The number of buffers in the Rx and Tx ring is set to 4096.

We use Click Modular Router version 1.7 for the implementation of the packet processing applications. Click runs on the Linux kernel version 2.6.24.7 in order to achieve high performance. We further rely on an Intel VTune Amplifier XE 2011 [6] for the CPU cycle measurement and analysis.

We implement and profile the following workloads:

- **Raw forwarding**: Packet forwarding without any layer-2 or layer-3 processing.

- **IPv4 forwarding**: IPv4 forwarding including IP lookup in a table with 380K entries, checksum computation, TTL decrement, and destination MAC address rewrite. Packets are generated with random destination IP addresses to stress the system.

- **AES encryption**: Packet encryption based on the Advanced Encryption Standard (AES).

- **CRC**: Checksum calculation for the whole packet payload.

Using our workload profiling technique, as exemplified in Section 3.2, we gauge the computational requirements of raw forwarding,

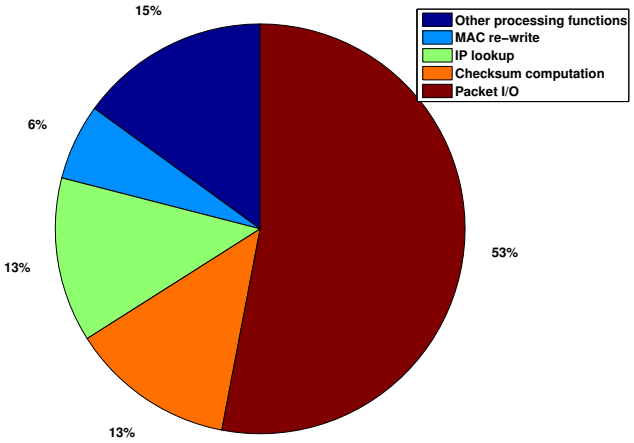


Figure 7: CPU cycles/sec distribution for IPv4.

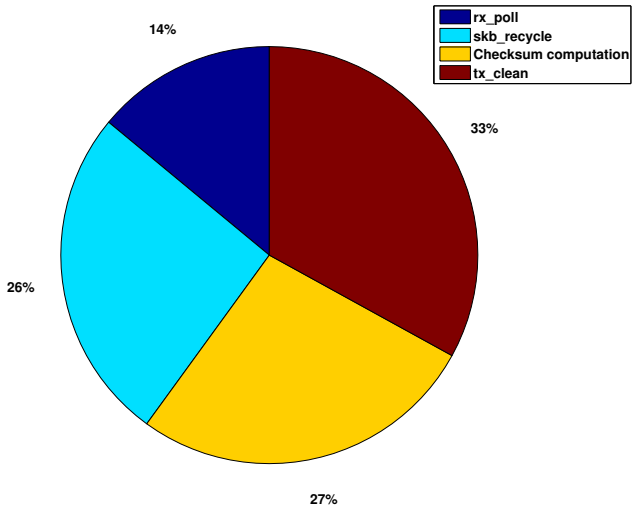


Figure 8: Main memory access distribution for IPv4.

IPv4, AES and CRC for diverse packet forwarding rates. Figure 6 illustrates the CPU cycles/sec versus the packet forwarding rate for raw forwarding, IPv4, and CRC with 64-byte packets. The standard deviation among different runs is negligible. As expected, raw forwarding incurs the lowest CPU utilization among all workloads since it basically comprises packet I/O handling along the receive and transmit path. In particular, packet I/O accounts roughly for 83% of the CPU cycles/sec consumed for raw forwarding according to our measurements. In comparison, IPv4 forwarding consumes more CPU cycles due to the additional processing operations performed on packets. In this respect, Figure 7 shows the distribution of CPU cycles for IPv4[1]. Approximately 32% of the measured CPU cycles are consumed for IP lookup, checksum computation and destination MAC address rewrite, resulting in an increased CPU utilization for IPv4 compared

1 The 15% of the CPU cycles/sec that correspond to "other processing functions" are consumed for packet enqueueing/dequeueing, timestamps, TTL decrement, Click counters and other operations with insignificant processing load.

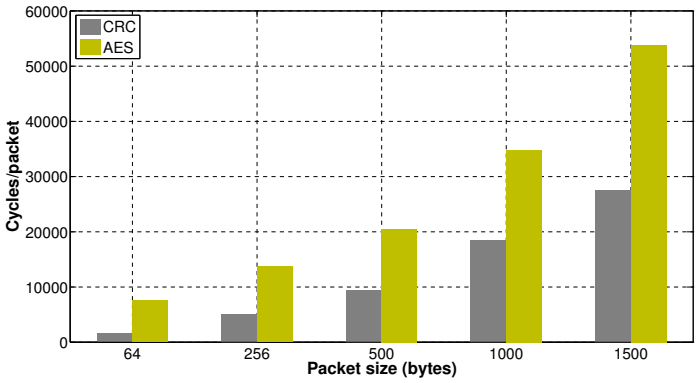


Figure 9: CPU cycles/packet vs. packet size for CRC and AES.

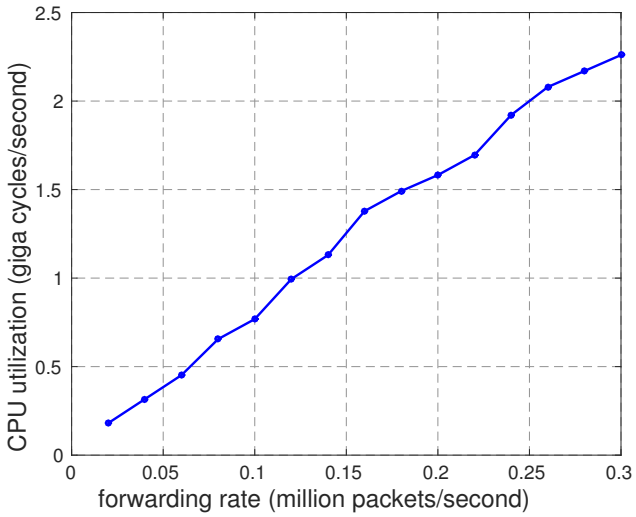


Figure 10: CPU cycles/sec for AES vs. packet forwarding rate with 64-byte packets.

to raw forwarding. Furthermore, Figure 8 illustrates the NIC driver and processing functions that incur main memory accesses for IPv4. Particularly, we measured approximately 3.79 main memory accesses per packet. Note that this number represents the main memory accesses initiated by the CPU (i.e., excluding the memory accesses initiated by the NIC). The large fraction of CPU cycles (i.e., 53% for IPv4) utilized for packet I/O indicates the significance of packet I/O handling for such workloads.

As shown in Figure 6, CRC yields relatively low computational requirements, due to the minimum packet size. In fact, CRC incurs less processing load compared to IPv4 for this packet length. However, measurements with larger packets (Figure 9) show that CRC requires a substantially larger number of CPU cycles per packet.

Figure 10 depicts the computational requirements of AES versus the packet forwarding rate with 64-byte packets. Figure 10 corroborates that AES is a CPU-intensive workload. Similar to CRC, AES

requires more CPU cycles per packet for larger packets, as shown in Figure 9.

Based on the measurements in Figures 6 and 10, Table 2 shows the average number of CPU cycles per packet for all four workloads with 64-byte packets. The study in [65], which also relies on a Click-based packet processing system without any advanced packet I/O handling, reports similar computational requirements for IPv4 (i.e., 1813 cycles/packet). These computational requirements can decrease (especially for workloads in which packet I/O accounts for a significant fraction of consumed cycles) when more efficient packet I/O techniques are applied [64], [86], [115]. In this respect, Table 3 depicts the average number of CPU cycles per packet for raw forwarding when the I/O optimizations of Netmap [115] and RouteBricks [64] are employed. Both I/O techniques transfer packets from NIC to memory in batches, while additional I/O optimizations of Netmap include static buffer allocation and the reduction of the data structure size for packet descriptors. Note that Click runs in user-space when Netmap is used. Compared to standard Click packet I/O handling, both Netmap and RouteBricks yield substantially lower computational requirements. This corroborates the strong impact of packet I/O handling on workload CPU utilization and manifests the need for efficient workload profiling methods.

So far, we discussed the computational requirements of packet processing applications that run alone on a single core. Depending on resource availability, a packet processing system can host multiple applications, performing the role of a multi-purpose software middlebox. In this context, we briefly discuss the impact of CPU resource sharing on the workloads' computational requirements. For workloads running on different cores (i.e., one workload per core), recent work [65] shows that the computational requirements of packet processing workloads increase due to L3 cache contention. The computational requirements of workloads sharing a single core may increase depending on scheduling and the size of their working sets. If their working sets do not fit into L3 cache, the additional main memory accesses will incur a performance penalty, increasing the required amount of CPU cycles to sustain a desired packet processing rate.

In terms of scheduling, multiple workloads can be assigned to a single (kernel) thread or to different (kernel) threads. The former requires a task scheduler so that multiple workloads can be executed within the same thread. Click provides such a scheduler, allowing multiple workloads to share a single core, without context switching. This does not incur a perceptible effect on the computational requirements of workloads. However, in this case, the migration of processing applications across cores is complicated and can result in traffic disruption. To facilitate migrations, a packet processing system can assign workloads (that share the same CPU core) to separate threads.

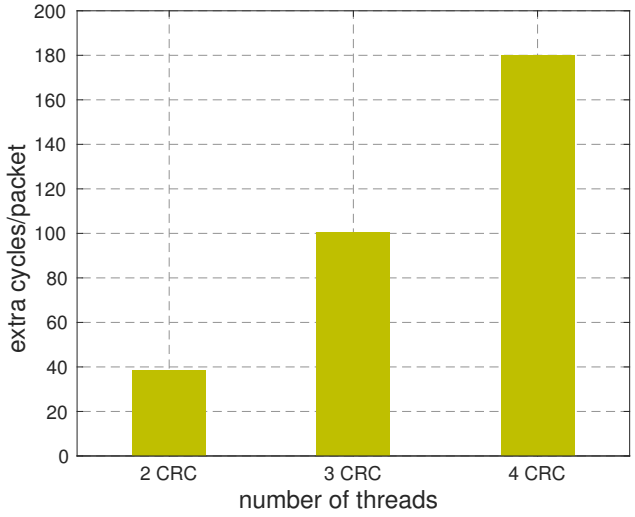


Figure 11: Extra CPU cycles/packet required for CRC with multi-threading.

Workload	cycles/packet
Raw forwarding	1415
IPv4	1874
AES	7931
CRC	1577

Table 2: Average CPU cycles/packet for different workloads (64-byte packets).

System	cycles/packet
Click	1415
Netmap	907
RouteBricks	926

Table 3: Average CPU cycles/packet for raw forwarding with diverse I/O optimizations (64-byte packets).

To investigate the processing requirements under multi-threading, we compare the CPU cycles required for CRC with single and multiple threads, running always on the same CPU core. Figure 11 depicts the extra cycles consumed with a diverse number of threads due to context switching. Note that this experiment does not incur any L3 cache contention since there is no application running on any of the other 3 cores.

3.4 RELATED WORK

The authors in [64] present Routebricks, a high performance software router architecture based on commodity servers and Click Modular Router. To identify the performance bottleneck, the authors briefly discuss a profiling method to estimate the computational requirements of a Click-based packet processing workload at a maximum forwarding rate by deducting the number of CPU cycles consumed for *empty polls*. In contrast, our profiling method measures CPU cycles at different forwarding rates and takes into account further workload-unrelated processing tasks (e.g., *empty dequeue*). We also consider the

effect of batch processing at low forwarding rates where fewer packets are available for polling. In summary, we measure the computational requirements of a packet processing workload by restricting the cycles entitled to the workload such that *empty polls* and *empty dequeues* are minimal, the number of packets per batch is maximum and the packet forwarding rate matches the target rate.

Several works have reported the computational requirements of various workloads. Packetshader [86] and Netmap [115] present an estimation of the computation overhead and the bottlenecks for packet I/O operations and subsequently propose optimization techniques to accelerate packet I/O processing. The work in [65] reports the computational characteristics of diverse Click-based workloads to study the performance predictability of software packet processing platforms under different assignments of workloads to system resources (e.g., CPU core, memory and cache). Furthermore, Kohler *et al.* [99] shows the results of measuring the processing time of different Click elements and a simple Click-based forwarding plane. In comparison to these works, in addition to reporting the computational requirements, we present a technique for the profiling of various packet processing workloads for a wide range of forwarding rates. Our profiling method is widely applicable to any packet processing system that uses polling and batch processing.

3.5 SUMMARY

In this chapter, we have exemplified methods to circumvent the difficulty of profiling packet processing workloads on commodity servers. We have discussed the implications of polling and batch processing on workload profiling and we further have shown that workload profiling can produce inaccurate results when the confluence of these two factors is not taken into account. Our experimental results demonstrate that our workload profiling technique can measure the computational requirements of various workloads for a wide range of packet forwarding rates. We believe that our workload profiling method can comprise a prominent component of a modern packet processing system, improving its ability to perform admission control and utilize the computing resources more efficiently.

4

INVARIANT PRESERVING MIDDLEBOX TRAVERSAL

The additional functionality that middleboxes embed in the network comes at a cost: middleboxes introduce various undesirable implications on traffic. For example, NATs rewrite IP addresses and ports, proxies break end-to-end semantics, and firewalls may block UDP traffic or cache out-of-order-packets introducing varying delays, while application optimizers can modify the packet payload [113, 130]. Furthermore, the deployment of firewalls and NATs along most Internet paths may hinder connection establishment with protocols such as SCTP [125] or Multi-Path TCP [131]. To mitigate these problems, most applications resort to tunneling, e.g., non-HTTP traffic may be tunnelled over HTTP to traverse firewalls; SCTP usually has to be tunnelled over TCP (or over UDP in case it is not blocked). Furthermore, traffic may be encrypted at the client device (e.g., using HTTPS) to inhibit payload modifications by application optimizers [113]. However, this increases power consumption in mobile devices.

Most of these implications stem from the middleboxes deployed by access ISPs and cellular networks. To obviate the need for tunneling or traffic encryption for middlebox traversal, we consider fostering the collaboration between end-hosts and ISPs. More precisely, an end-host can express its requirements for the establishment of a certain type of connection, e.g., do not modify packet fields or payload, permit UDP traffic or access to public DNS servers. Such requirements can be specified in the form of invariants (e.g., using the API in [113]). Upon the submission of such a request, the ISP may be willing to redirect the traffic through a set of middleboxes (e.g., NAT and firewall) that comply with its security policy and, at the same time, preserve the invariant expressed by the end-host. This can be offered to ISP clients as a value-added service, which may be appealing to a wide range of users (e.g., home network users, mobile users, enterprises) that currently experience limitations in the applications or services they can run.

Establishing connections through a sequence of invariant preserving middleboxes raises several requirements: (i) the collection of middlebox configurations, (ii) parsing and checking middlebox configurations against requested invariants, (iii) the selection of invariant-preserving middleboxes and shortest paths, and (iv) the insertion of forwarding entries in the ISP's routers to route the traffic through the assigned path. To this end, we present an SDN architecture for invariant preserving middlebox traversal. Following the trend for

(logically) centralized control, we rely on a centralized controller deployed by the ISP, which retrieves middlebox configurations, selects middleboxes and paths that preserve the specified invariant, and sets up packet forwarding along the selected path. Middlebox checking against invariants can be performed using recent advances on static analysis, such as HSA [96] or SymNet [128]. For the installation of flow entries to routers, we employ OpenFlow [107]. Our work is mainly focused on middlebox and path selection. To this end, we present and evaluate an algorithm for the selection of a path through a set of invariant-preserving middleboxes. Our simulation results show that our approach substantially increases the number of established connections, especially under low and moderate levels of network utilization.

The remainder of this chapter is organized as follows. In Section 4.1, we review and discuss the different implications of middleboxes on traffic and connection establishment. Section 4.2 provides an overview of our SDN architecture. In Section 4.3, we discuss our algorithm for invariant preserving path selection. In Section 4.4, we present our simulation environment and results. Finally, in Section 4.6 we highlight our conclusions.

4.1 MIDDLEBOX IMPLICATIONS

In this section, we discuss the implications of widely used middleboxes on traffic and connection establishment:

- **NATs:** Due to the limited size of IPv4 address space, NATs have become one of most popular middleboxes on ISP networks, especially on cellular networks [130]. They enable the sharing of a public IP address among multiple hosts with private IP addresses by mapping the private IP address and the source port number of a host connection (TCP or UDP) to the public IP address and a selected port number. As a result, hosts sitting behind NATs are not visible to the outside world, i.e., establishing connections with NATed hosts (e.g., a VoIP, Peer to Peer (P2P) applications) requires complicated NAT traversal techniques [83, 70, 44] and might need the participation of a third party (e.g., a relay [98]). However, even with NAT traversal techniques, connection establishment could fail due to the ISPs' policies and configurations. As shown in [130], to perform load balancing cellular network operators may assign multiple NATs to a single device. Subsequently, this hinders NAT traversal techniques that depend on learning the NAT's public IP address by establishing multiple connections with the NAT (since different connections are handled by different NATs). Furthermore, operators might configure NATs to assign random port numbers to

the mapped connections which hampers applications (e.g., P2P) performing NAT traversal by trying to infer the mapped NAT port number [130].

- **Firewalls:** Firewalls are essential to today's network functionality by providing protection against malicious traffic as well as untrusted and policy-violating accesses. However, despite their importance, firewalls have become an obstacle hindering not only the deployment of new protocols and extensions (e.g., SCTP [125], ECN [114]), but also restricting the connectivity of the traditional protocols. More particular, recent studies have shown that applications and protocols are being forced to tunnel over HTTP/HTTPs to bypass firewalls [113] [130]. Even when connections are successfully established, firewalls introduce further implications such as buffering out-of-order packets, which impairs the functionality and performance of TCP connections, and terminating long-lived flows due to short timeouts on firewalls, which leads to increased power consumption and service disruption for end hosts[130].

- **Proxies:** Proxies perform several functions to optimize the performance of particular applications or protocols such as caching contents, data compression and TCP connections splitting. Proxies are usually implemented with a specific application in mind which impairs the functionality of new applications passing through them. For example, mobile devices have to tunnel over HTTPS to avoid HTTP optimizers breaking their protocol semantics by modifying their packets payload or by sending a cached reply instead of forwarding the packet to the end server [113].

4.2 ARCHITECTURE OVERVIEW

In this section, we discuss the requirements of invariant preserving middlebox traversal and present the components of our SDN architecture.

Consider the example in Figure 12, where a trusted user (e.g., an enterprise with well-established relation/contract with the ISP) is trying to access a server through port 1443. Since the firewall on its default path (path 1) blocks any traffic on ports other than port 80, the user fails to establish a connection with the server. A straightforward solution is to request the ISP to reconfigure the firewall on the default path. However this will allow the malicious user's traffic to traverse the network. An alternative solution, which we consider in this work, is to allow the user to express her requirement as an invariant to the

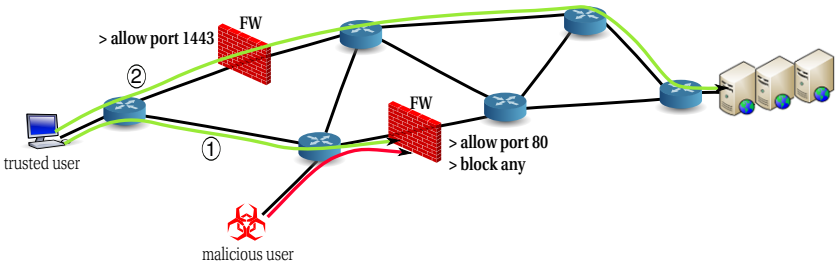


Figure 12: Example of invariant preserving middleboxes traversal.

ISP (in this case allow port 1443), and in turn, the ISP identifies a path which preserves the invariant and does not violate the the ISP policy (path 2).

To this end, we envision an SDN architecture where a centralized control plane provides invariant preserving routing and redirection through the ISP network. Accordingly, we assume the deployment of OpenFlow switches which serve as the data plane of the network. Furthermore, as in today's ISP network, a set of middleboxes are deployed in the network at different locations to provide services such as protection against malicious traffic (e.g., firewalls), enable the sharing of IPv4 addresses (NAT) and caching of frequently used content (proxies). To provide invariant preserving connection establishment, our SDN architecture needs to fulfill a set of requirements:

- **Efficient resource management:** We consider two objectives for resource management in ISP networks: (i) delay minimization, where an ISP aims at routing traffic through the path with the lowest delay and (ii) load balancing, where an ISP aims at balancing the traffic load across the network.

- **Correctness:** Traffic should traverse paths that preserve the invariant while not violating the ISP policy, e.g., a video flow with a particular port number might be redirected through a firewall which grants access to it but still needs to keep an upper bound on the BW consumed by this flow. This requires correct and efficient parsing and checking of the state and configuration of the middleboxes deployed in the network.

- **Traffic redirection:** The controller should be able to install forwarding entries in the ISP's switches to reroute traffic through the selected path. This should also take into account middleboxes which modify the packets routing header fields such as the IP addresses (e.g., NATs, load balancer).

To fulfill these requirements, we design a control plane which consists of four components (Figure 13):

- **MBs configuration and state collection**: This component collects and stores the state and configurations (e.g., firewall rules) of each middlebox deployed in the network. It accesses middleboxes through interfaces exposed to the controllers by the vendors. These interfaces could be vendor-specific (e.g., CISCO CLI) or standard interface such as NetConf [69] or SIMCO [126].

- **Static checking**: This component implements tools such as SymNet [128] or HSA [96] to parse and analyse the state and configurations of middleboxes against the requested invariants. It basically identifies the implications the middleboxes have on the flow and hence, specifies the middleboxes which do not violate the flow invariant.

- **Network monitoring:** This component keeps track of the network topology as well as the network links and middleboxes utilization. It reads the counters of the network switches deployed on the network using OpenFlow [107]. For monitoring middleboxes utilization, it uses again the interface exposed by the vendors.

- **Path selection:** Based on the output provided by the static checking and the network monitoring component, this component selects a path which fulfils the invariant of the connection while considering the utilization of the network and the middleboxes. It implements our path selection algorithm presented in Section 4.3.

- **Switches configuration:** This component installs the required flow entries in OpenFlow switches to redirect the flow through the path selected by the path selection component. For middleboxes (e.g., NAT) which modify some of the flow's 5 tuples (source and destination IP addresses, source and destination port numbers, protocol), the flow can be identified by adding tags to each packet such as in [76] and [72].

4.3 PATH SELECTION

We develop an algorithm which selects a network path traversing middleboxes that preserves a connection invariant. In addition to the

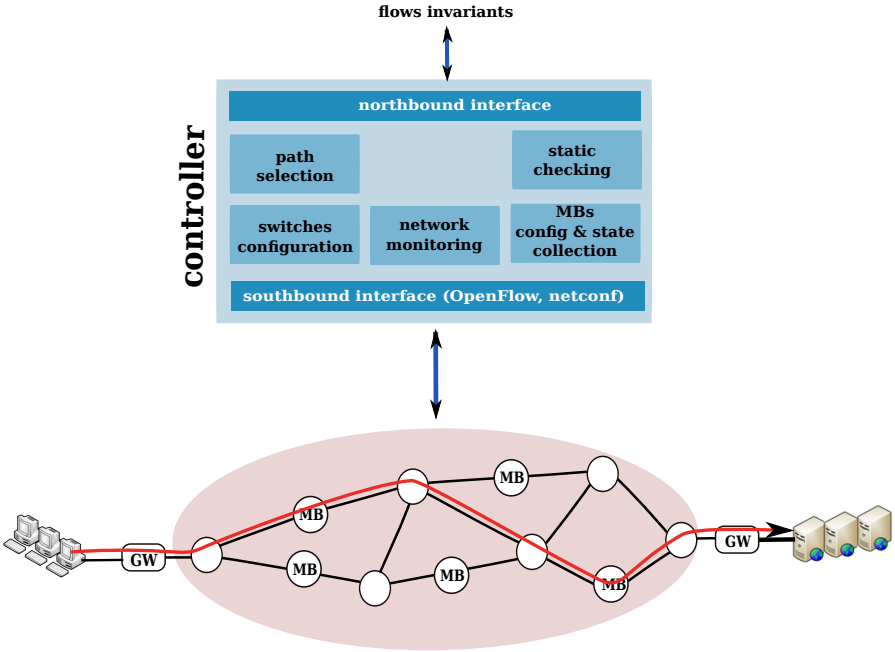


Figure 13: Architecture components.

configuration and the state of each middlebox, our algorithm takes into account the available bandwidth on each network link as well as the available processing capacity of each middlebox. The algorithm is executed by an SDN controller which has the knowledge of the network topology and utilization, the middleboxes utilization and state, and the connection invariant. We present two variants of our algorithm: the first aims at minimizing end-to-end delay, whereas the second strives to achieve load balancing across the network.

We represent the ISP network as a weighted undirected graph $G = (N, L)$, where N is the set of nodes and L is the set of links between nodes of the set N. Nodes are classified into a set of routers R and a set of middleboxes M such that $N = R \cup M$. Each m_i has a processing capacity which is denoted by $CP(m_i)$ and a state $S(m_i)$. Each link $l_{ij} \in L$ between two nodes n_i and n_j is associated with the available bandwidth $C(l_{ij})$. Let P_{ij} represents the set of paths in the network G, between the pair of nodes n_i and n_j. The available bandwidth $C(p)$ of a path $p \in P_{ij}$ is given by the minimum residual bandwidth of the links along the path:

$$C(p) = \min_{l_{ij} \in p} C(l_{ij}) \tag{1}$$

We further represent a connection demand with a vector $d = \{n_{src}, n_{dst}, r, cmp, v\}$, where $n_{src}, n_{dst} \in N$ denote the connection source and destination nodes, r represents the traffic rate, cmp is the required computing capacity to process the traffic, and v is the connection invariant.

Algorithm 2 Path selection

Inputs: $G = (N, L), d$

for each $l \in L$ **do**
  if $C(l) < r$ **then**
    delete l from G
  end if
end for

$P_{ij} \leftarrow$ FIND_ALL_PATHS(n_{src}, n_{dst}) // all paths between source and destination
SORT(Pi, j) // sort paths based on their length or available BW

for each $p \in P_{i,j}$ **do**
  $found \leftarrow true$
  for each $m \in P$
    if $cmp > CP(m)$ **or** $v \cap S(m) = \emptyset$ **then**
      $Found \leftarrow false$
      break
    end if
  end for
  if $found$ **then**
    return p
  end if
end for
return $\emptyset$ // no path was found

The algorithm (Algorithm 2) selects a path between the source and the destination of a connection. It starts by removing all the links with insufficient available bandwidth to fulfil the rate of the connection. This step reduces the size of the graph that the FIND_ALL_-PATHS function has to process to calculate all paths between the source and the destination. This function implements the algorithm in [118] which has a complexity of $O(|N| + |L|)$ for a single path. Reducing the size of N and L results in lower runtime. After identifying all the paths, we sort them in increasing order based on the number of hops or on a decreasing order based on the available bandwidth $C(p)$. Sorting based on the number of hops results in delay minimization, whereas sorting based on the available bandwidth achieves load balancing. We term the two algorithm variants as *invariant preserving SP algorithm* and *invariant preserving LB algorithm*, respectively. Finally, the algorithm goes through the sorted paths to find the first one which fulfills the invariant as well as the connection processing demand.

4.4 EVALUATION

In this section, we evaluate the efficiency of our path selection algorithms for invariant preserving middlebox traversal. In particular, we

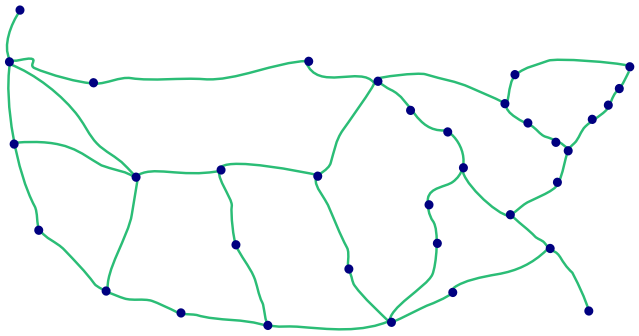


Figure 14: Simulation OpenFlow switches topology.

use simulation to measure the connection establishment rate and the network and middleboxes utilization. Furthermore, we compare our algorithms in terms of load balancing level and path hop counts per connection.

4.4.1 *Evaluation Environment*

We have developed a Python flow-level simulator to establish invariant preserving connections through an ISP network. To model the ISP network, we use Internet2 topology [3] which consists of 34 nodes (Figure 14). Each node in this figure represents an OpenFlow switch, whereas each edge is a network link with 1 Gbit/second bandwidth. At different locations of the topology, we deploy 12 middleboxes. Each middlebox has 10 GHz CPU capacity and performs access control using a randomly generated list of destination port numbers (each middlebox works as a stateless firewall). We generate non-expiring connections with destination port numbers, rates, and processing demands sampled out of a uniform distribution. For each generated connection, we randomly select a source and a destination switch. Using the algorithm in Section 4.3, a connection is established if a network path which preserves the connection invariant and fulfills its rate and processing demand is found; otherwise, it is rejected. For each successfully established connection, the bandwidth of the links and the processing capacity of middleboxes on the path are updated accordingly.

We compare the efficiency of our approach against the traditional shortest path selection. In particular, for each new connection we calculate the shortest path between the connection source and destination. If the shortest path fulfils the connection demand and invariant, the connection is established; otherwise, it is rejected.

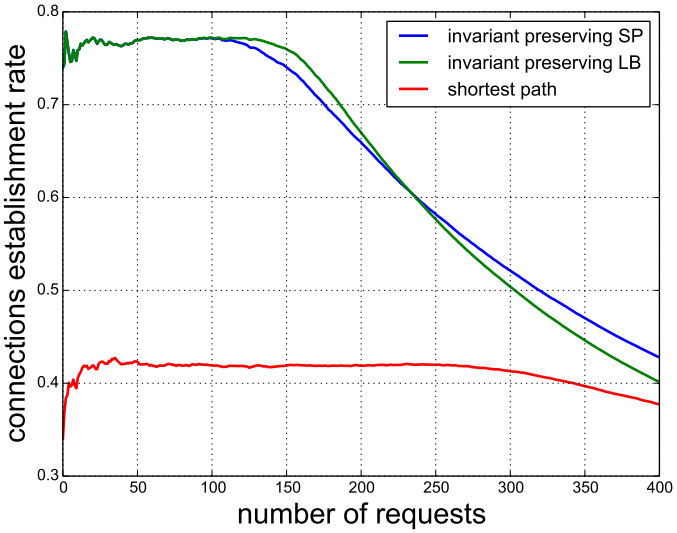


Figure 15: Connection establishment rate vs. number of arriving requests.

We conducted our simulations on a machine with an Intel Core i5 quad-core CPU at 3.20 GHz and 16 GB of RAM. We repeated each experiment 100 times and report the average.

4.4.2 *Evaluation Results*

We start by measuring the *connection acceptance rate*. This represents the percentage of connections' sizes for which invariant-preserving paths were selected. As Figure 15 shows, our algorithms (invariant-preserving SP and invariant-preserving LB) establish almost 40% more connections than the traditional shortest path selection. This is because our algorithms select alternative paths when either the invariant or the connection demand are not fulfilled, whereas the traditional shortest path rejects the connection when the shortest path does not meet the connection requirements. This can be also seen through the evolution of network and middleboxes utilization (Figure 17 and Figure 16). It also illustrates that the invariant-preserving LB algorithm achieves better utilization and reaches saturation faster than the invariant-preserving SP. This is because the invariant-preserving LB accepts more requests at the beginning when the network resources are underutilized.

We also measure the network *load balancing level* which we define as the maximum link utilization over the average link utilization across the ISP network. Lower values of the load balancing level represent better load balancing, whereas a value of 1 designates optimal load balancing. As we can see in Figure 18, the invariant-preserving LB algorithm outperforms both our invariant-preserving SP algorithm and the traditional shortest path.

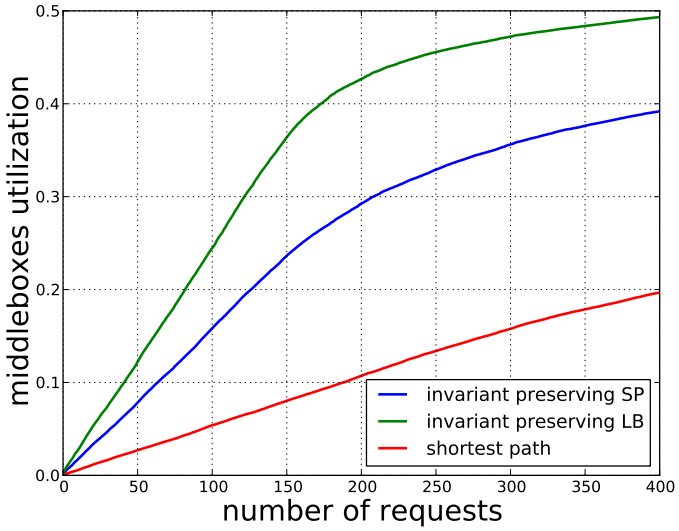


Figure 16: Total utilization of middleboxes deployed on the network.

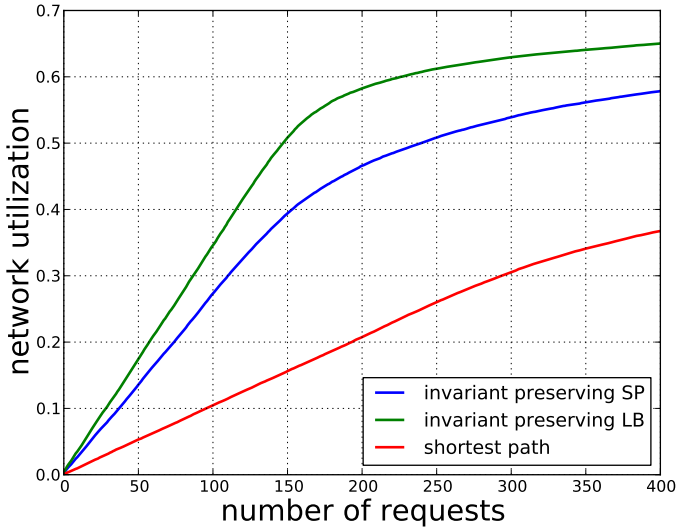


Figure 17: Total utilization of all network links.

We further look at the path length per selected path in terms of the number of hops. As we expect, the invariant-preserving SP outperforms the invariant-preserving LB algorithm (Figure 19); however, as the network resources become more utilized the difference between both algorithms diminishes. This is because the number of alternative paths with sufficient capacity decreases which limits the solution search space for both algorithms.

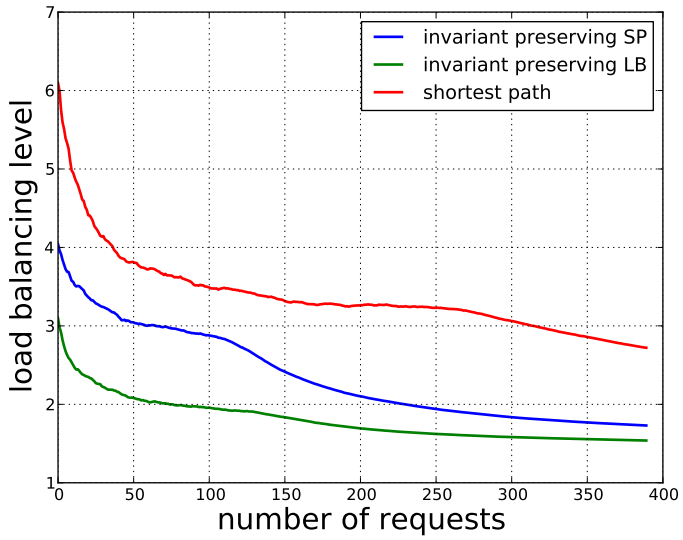


Figure 18: The network load balancing level.

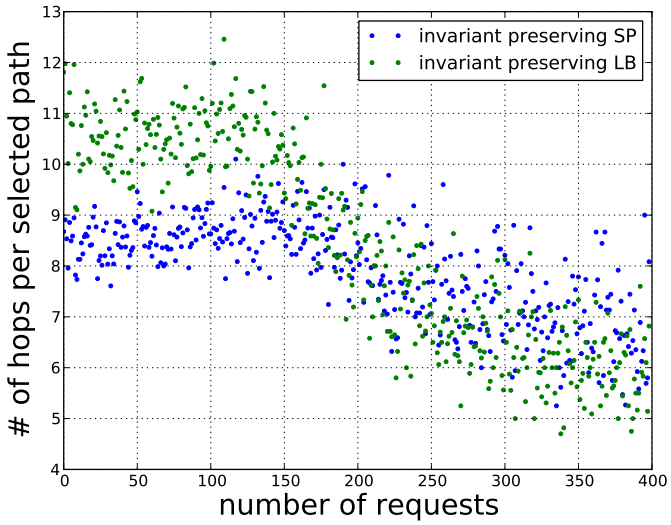


Figure 19: The length of each selected path for each connection.

4.5 RELATED WORK

Several signaling protocols have been proposed for middlebox configuration and traversal. In particular, protocols such as SIMCO [126], NAT/firewall NSIS [127], STUN [116] and NUTSS [83] enable middlebox traversal by allowing end points to dynamically configure NATs/firewalls deployed on the traffic path. However, these protocols are device and flow specific and do not consider the global policy of the ISP network, i.e., a traversal ordered by a trusted end point might expose the ISP network to the malicious traffic of another end point. In contrast, we propose an SDN architecture where a centralized controller with a network-wide visibility provides invariant preserving middleboxes traversal while taking into account the ISP global policy and the available resources (i.e., middlebox and network utilization).

On the other hand, OpenNF [80] and [79] develop a southbound interface to enable a centralized controller to manipulate the state and configuration of a wide range of middleboxes. Furthermore, a recent work [113] proposes an API to allow end points to express their invariants to a network operator which, in response, informs the end point whether the network preserves or violates this invariant. These works are complementary to our SDN architecture providing the means for collaboration between the end point and the ISPs and to collect middleboxes' state and configurations.

Flows redirection and assignment is considered by several works to optimize middleboxes resources utilization. For instance, SIMPLE [111] proposes an SDN-based architecture to assign flows to middleboxes such that middlebox policy is enforced and load balancing is achieved. The work in [89] employs flows assignment and redirection to balance IDS load across multiple nodes using a centralized controller with a network-wide view. Similar to these works, we rely on flow redirection and assignment to provide invariant preserving middlebox traversal while considering load balancing and delay minimization for resource allocation.

4.6 SUMMARY

In this chapter, we presented an SDN architecture for establishing invariant preserving connections traversing middleboxes and fostering the collaboration between end-hosts and ISPs. In particular, an end-host can express a desirable behaviour from the network, specified as an invariant (e.g., no IP header or payload modification), and the ISP, in turn, can establish a connection through middleboxes that preserve this invariant. To this end, we have developed an algorithm to select redirection paths through a sequence of invariant-preserving middleboxes while considering network and middlebox utilization.

Our algorithm can be adapted to fulfill different objectives: load balancing or delay minimization.

Using simulations, we have shown that our algorithm substantially increases the number (more than 40%) of established connections with invariant preservation and achieves a network-wide load balance as well as high network and middleboxes utilization. We have further measured the path length for each established connection and have shown that based on the ISP policies different variants of our algorithm can be used.

5

MIDAS: MIDDLEBOX DISCOVERY AND SELECTION FOR ON-PATH FLOW PROCESSING

As discussed in the introduction, outsourcing NF processing to the network can facilitate the deployment of new services and is essentially a prerequisite for emerging communication paradigms such as information-centric networking [93]. Network processing further enables new business models such as NFaaS, leading to a significant reduction in operational and technology investment costs in enterprise networks. Moreover, supported by the growing tendency to deploy programmable and general purpose processors on routers and switches [12, 20, 8, 97] and the increasing number of micro-datacenters deployed at ISPs' PoPs [78], NF processing can be provided along the flow path obviating the need for traffic redirection which can lead to latency inflation. However, the deployment of a network service chain requires the assignment of NFs to middleboxes, while preserving correctness and satisfying any NF location dependencies. The latter, in turn, will create the need for multi-provider NF assignment, when the geographic footprint of NFPs does not satisfy such location constraints. The presence of multiple NFPs exacerbates the NF assignment problem, since NFP policies (that will be likely governed by current ISPs' policies) will enforce significant restrictions on information disclosure (i.e., network topology, resource availability) and interoperability with third parties.

A prerequisite for NF assignment is the discovery of the middleboxes deployed along the traffic path. Existing techniques for path discovery (e.g., *traceroute*) and recent extensions for middlebox detection (Tracebox [59]) incur delays that will substantially prolong flow processing establishment. Protocols for middlebox signaling have been developed primarily for device configuration and are applicable only to certain types of middleboxes (e.g., NAT and firewall configuration with SIMCO [126] and NSIS [87]). Furthermore, AS-path retrieval from BGP routers will not augment middlebox discovery since the sequence of middleboxes traversed by the flow within each NFP will still be required.

To mitigate the problems of middlebox discovery and selection for on-path processing, in this chapter we present MIDAS, an architecture for the coordination of processing setup using a centralized middlebox controller in each NFP. MIDAS comprises protocols for middlebox signaling, controller chaining, and methods for middlebox selection across multiple NFPs. We employ MPC [57, 45] for middlebox selection in order to preserve the confidentiality of middlebox utiliza-

tion across the NFPs. Using a prototype implementation, we study the feasibility of MIDAS by measuring the delays incurred during the flow processing setup with a diverse range of middleboxes, NFPs, and network service request arrival rates. We further use simulations to assess the efficiency of our middlebox selection approach in terms of load balancing and request acceptance.

The remainder of this chapter is organized as follows. Section 5.1 describes the middlebox discovery and selection problems. Section 5.2 provides an overview of MIDAS. In Section 5.3, we present our protocols for middlebox signaling and controller chaining. In Section 5.4, we introduce our approach for middlebox selection across multiple NFPs. Section 5.5 describes the implementation of our consolidated middlebox and signaling protocols. In Section 5.6, we discuss our experimental and simulation results. Section 5.7 reviews the related work. Finally, Section 5.8 highlights our conclusions.

5.1 CHALLENGES AND REQUIREMENTS

In this section, we discuss the requirements and challenges of middlebox discovery and selection for on-path processing establishment across multiple NFPs. We envision the migration of NFs from enterprise networks to NFPs (e.g., micro-datacenter network operators), as exemplified by NFV. We particularly consider NFPs deploying Consolidated Middleboxess (CoMBs) on commodity servers, routers with general purpose processors [12, 20, 8, 97] or programmable hardware devices (e.g., NetFPGA [19]). In this respect, CoMBs enable the selective deployment of NFs using software components [106, 119].

We rely on service chaining for the representation of network services (Section 2.2). Service chains may contain location-dependent NFs, such as Redundancy Elimination (RE) or proxies which should be in proximity to the client (e.g., enterprise network) to reduce latency and conserve bandwidth. On the other hand, packet filters or IDS should be deployed near the traffic sources for improved efficiency against Denial-of-Service attacks. Service chains containing NFs with location dependencies may require a large provider footprint, i.e., PoPs near both connection end-points. ISPs and large cloud providers (e.g., Amazon) may not meet these location constraints, especially for clients residing outside of US. Therefore, we tackle the problem of NF assignment across multiple administratively independent NFPs.

Figure 20 illustrates the problem of assigning the NFs composing a service chain onto CoMBs hosted by multiple NFPs. In this example, RE, cache, and web application firewall are location dependent, and thereby, should be assigned to middleboxes hosted by NFP 1 and NFP 3. On the other hand, flow monitoring and load balancing do not yield any location restrictions, and, as such, their assignment is

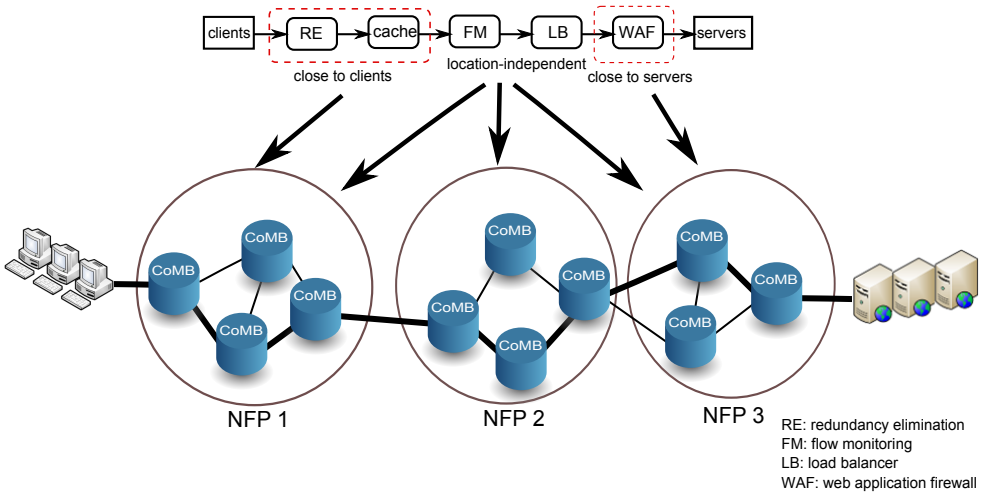


Figure 20: Assignment of NFs to middleboxes along the traffic path.

subject to NFP policies, resource availability, and other constraints such as preserving the correct order of NFs.

We particularly consider the following objectives for on-path processing establishment:

Performance. Flow processing setup should incur low delay, while forwarding rates should be high and comparable with middleboxes built of specialized hardware. Since the latter has been addressed by recent work (e.g., [106, 115]), we focus on achieving low processing setup delays.

Efficiency. Network-wide load balancing has been considered as a desired property for the deployment of network processing elements [75] and IDSs [89], and, as such, we aim at equalizing the processing load among CoMBs.

Correctness. NFs assignment should preserve the order specified in the service chain.

These objectives essentially designate CoMB selection as a key challenge for on-path processing. The CoMB selection problem is exacerbated by NFP policies that restrict the information disclosure and interoperability with third parties. Specifically, CoMB utilizations will be deemed confidential and, thereby, will not be disclosed by NFPs. This essentially hinders collective decisions among NFPs for NF assignment.

A simple approach to CoMB selection consists in allowing CoMBs to autonomously pick and process incoming traffic. More precisely, CoMBs can select a flow depending on its current level of utilization (e.g., with a probability that is inversely proportional to the CoMB utilization [75]) and, subsequently, encode the processing status into the packet header to prevent any processing duplication from down-

stream CoMBs. Such on-the-fly processing obviates the need for NFP interoperability and also does not require maintaining any state besides the flow processing status. However, this approach does not guarantee correctness since it is very difficult to embed a sequence of NFs without any knowledge about the number and utilization of downstream CoMBs. Possible ways to mitigate this, i.e., seeking the co-location of all NFs or alternatively performing a first-fit for each NF, could result in load imbalance among CoMBs, violating the efficiency property.

To meet all of these objectives, NF assignment requires the knowledge of the CoMBs deployed along the traffic path. Since the time required for path discovery with *traceroute* and its recent extensions (i.e., Tracebox [59]) is prohibitive for on-path processing establishment, we present protocols and algorithms for middlebox discovery and selection, while adhering to NFP policies and preserving the confidentiality of CoMB utilization across NFPs.

5.2 ARCHITECTURE OVERVIEW

Hereby, we present MIDAS, an architecture that (i) enables interoperability across multiple NFPs for the discovery of CoMBs along the traffic path and (ii) fosters the collaboration between NFPs for CoMB selection without disclosing any confidential information. MIDAS mainly relies on three components (Figure 21): (i) the CoMBs, (ii) a logically centralized CoMB controller residing in each NFP, and (iii) the Network Processing Client (NPCL) which submits the network service request on behalf of the client.

MIDAS supports NPCL located on-path (e.g., end-host, access gateway, edge router) as well as off-path (e.g., network management system, access control server). For off-path NPCL, we mitigate the potential problem of different gateways assigned to the NPCL and to the end-host(s) (for which the network service is being set up) by redirecting any middlebox signaling messages (e.g., using a tunnel) through the gateway assigned to the end-host(s). This ensures that signaling messages and data traffic will traverse the same CoMBs, assuming that any load balancing by routers is per flow or per destination.

On-path processing is established on a per-flow basis, as a sequence of the following steps:

Middlebox Signaling. MIDAS uses a signaling protocol for the discovery of CoMBs along the data path. As soon as a CoMB has been identified, the CoMB sends a notification to its controller. We elaborate on CoMB signaling in Section 5.3.1.

Controller Chaining. MIDAS establishes a chain between the controllers of the discovered CoMBs. This enables the controllers to make collective decisions for CoMB selection. Controller chaining is discussed in Section 5.3.2.

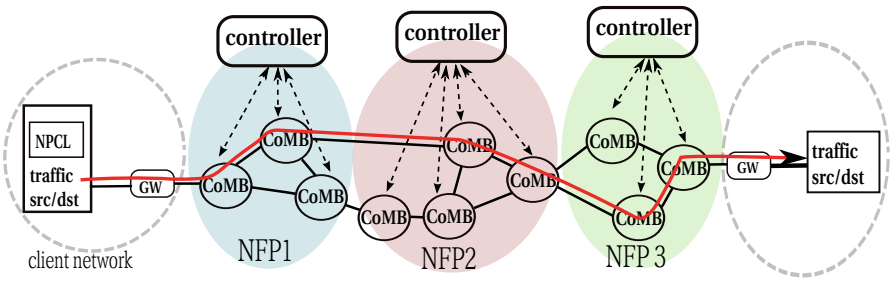


Figure 21: Architecture components.

Middlebox Selection. Middlebox selection consists of (i) CoMB selection within each NFP and (ii) the assignment of NFPs via the collaboration of their controllers. Intra-provider CoMB selection is carried out using a heuristic algorithm that strives to achieve load balancing across the CoMBs, while preserving correctness. To preserve confidentiality for NFP assignment, we leverage on Multi-Party Computation (MPC) [57, 45]. Despite being computationally intensive, MPC has been considered as a viable solution for computations with privacy concerns and has found applications in network monitoring [63] and inter-domain routing [85, 88]. Further details on CoMB selection are given in Section 5.4.

Processing Module Instantiation. Upon the CoMB selection, the controller instructs the assigned CoMB(s) to install and configure the required Processing Modules (PMs). This essentially comprises resource allocation, PM configuration (e.g., *rules installation*), PM installation, and configuration of packet steering between the physical ports and the PM's virtual interfaces. We have implemented such a CoMB on a commodity server. The functionality and implementation of the main CoMB components are discussed in Section 5.5.

5.3 MIDDLEBOX DISCOVERY

Middlebox discovery consists of (i) middlebox signaling (Section 5.3.1), i.e., identifying the CoMBs along the traffic path and sending notifications to their controllers, and (ii) controller chaining (Section 5.3.2).

5.3.1 *Middlebox Signaling*

MIDAS initiates middlebox discovery by signaling all CoMBs along the data path. To this end, the NPCL generates a *REQUEST* message which traverses all downstream CoMBs, as shown in Figure 5.22(a). The *REQUEST* message contains the following fields (Figure 5.23(a)):

- *Service Request ID (SRID)*, whose scope is limited to the client. However, concatenating the SRID and the client's IP address

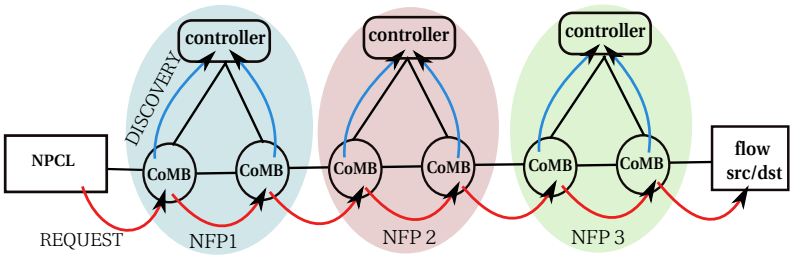
allows the generation of globally unique IDs without the need for any coordination between clients. To reduce the length of the SRID field, we reuse the SRIDs of expired requests.

- *Controller address*, which is used to store the IP address of the controller of the current or the upstream NFP, as explained below.
- *CoMB address*, which carries the IP address of the upstream CoMB.
- *Processing type identification*, which corresponds to a specific NF (e.g., IDS) advertised by an NFP. A service chain will contain the processing type ID of each NF.
- *Proximity*, which carries the distance (specified in km, milliseconds or number of hops) to traffic source or destination for each NF identified in the *Processing type identification* field.
- *Traffic rate*, which is used to derive the NF computational requirements (e.g., the CPU cycles can be calculated using cycles/packet measurements obtained with our profiling method in chapter 3) and, subsequently, examine the feasibility of NF-to-CoMB assignment.
- *Flow specification*, typically expressed with the 5 tuple (source/destination IP address, source/destination port number, and protocol) but can be also extended to cover L4+ protocols such as HTTP. Flow specification can include wild-card entries.

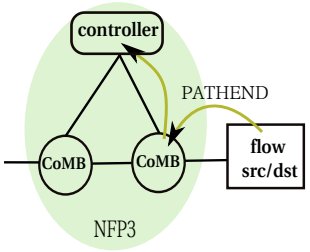
Upon receiving the REQUEST message, each CoMB replaces the *CoMB address* field with its own IP address and compares the *Controller address* field with the IP address of its own controller. If both addresses match, the CoMB simply forwards the REQUEST message without further modifications. Otherwise, the CoMB stores the controller address of the *REQUEST* message and inserts the IP address of its own controller in the *Controller address* field of the message. The *Controller address* update essentially takes place in the first CoMB encountered by the *REQUEST* message in each NFP. This enables each controller to learn the address of its upstream controller, augmenting controller chaining.

Upon signaling, each CoMB announces its discovery and conveys its utilization to its controller, using a *DISCOVERY* message as depicted in Figure 5.22(a). The *DISCOVERY* message contains the CoMB identifier and utilization, as well as a copy of the *REQUEST* message (Figure 5.23(b)).

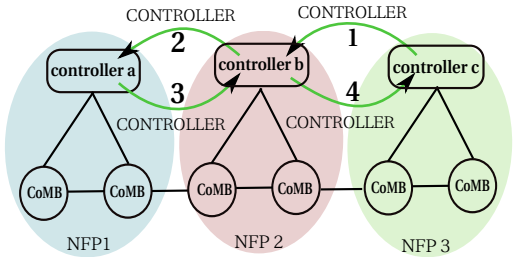
MIDAS uses a *PATHEND* message to bind middlebox signaling with controller chaining. This message is sent by the other end-point



(a) Middlebox signaling



(b) Notification of data path end



(c) Controller chaining

Figure 22: Middlebox discovery steps.

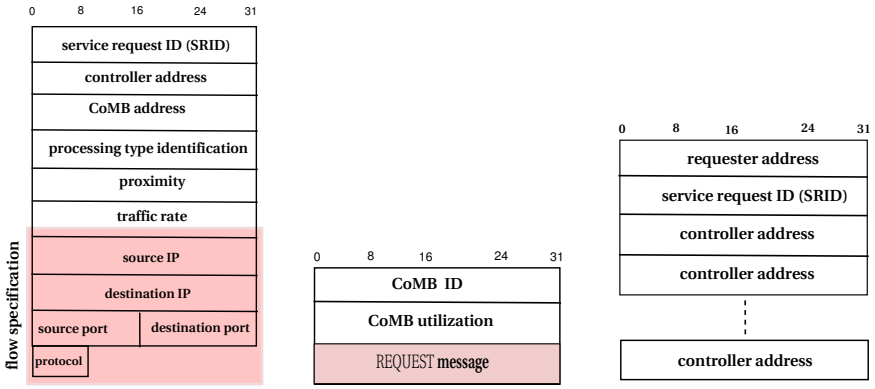
to the last CoMB (using the *CoMB address* field in the *REQUEST* message) and is subsequently relayed to the CoMB's controller, initiating controller chaining (Figure 5.22(b)). The *PATHEND* message contains only the SRID and the IP address of NPCL, since the controller can easily retrieve the request specification (i.e., appended in the *REQUEST* message) by looking up the SRID in all messages sent by each discovered CoMB.

5.3.2 *Controller Chaining*

Controller chaining is an iterative process, in which all controllers exchange their IP addresses[1] so that they can collectively make decisions for CoMB selection.

Controller chaining is triggered as soon as a *PATHEND* message has been received by the controller of the last NFP on the data path. This controller first extracts the SRID and the NPCL's IP address from the *PATHEND* message and subsequently performs a lookup on the *DISCOVERY* messages received in order to retrieve the IP address of the upstream controller. Next, the controller appends its IP address, SRID and NPCL's IP address to a *CONTROLLER* message (Figure 5.23(c)), which is sent to its upstream controller (Figure 5.22(c)). Each

1 For security reasons, a controller may announce the IP address of a proxy instead of its own IP address.



(a) *REQUEST* message format

(b) *DISCOVERY* message format

(c) *CONTROLLER* message format

Figure 23: Message formats.

controller receiving a *CONTROLLER* message appends its own IP address and forwards the message to its own upstream controller. When the first controller has been reached, this procedure is repeated over the reverse path, i.e., controller addresses are appended to *CONTROLLER* messages which are being forwarded downstream, until controller chaining has been completed.

5.4 MIDDLEBOX SELECTION

Middlebox discovery is followed by the assignment of NFs to CoMBs hosted by multiple providers. This essentially requires the assignment of NFs to NFPs (among the ones discovered along the traffic path) and the selection of the CoMBs within each assigned NFP. In this respect, we employ MPC for NFP assignment and further use a heuristic algorithm for the CoMB selection. Since MPC incurs higher computational and communication overhead than our heuristic algorithm, we first use our heuristic to examine the feasibility of NF-to-CoMB assignment within each NFP and, subsequently, execute MPC only across the NFPs in which NF assignment is feasible.

5.4.1 *Intra-Provider Middlebox Selection*

CoMB selection within each NFP is carried out by the NFP's controller which is assumed to have up-to-date information about the CoMB utilizations. The difficulty in CoMB selection stems from the need to ensure correctness for service chains. To this end, we present a heuristic algorithm for the assignment of NFs to CoMBs in the correct order. The objective of the algorithm is to achieve load balancing across the CoMBs.

Let R denote a network processing service request, consisting of N NFs, each one associated with a processing demand represented by $d_i, i = 1, ..., N$. We also define the residual capacity of each CoMB j deployed along the flow's path as $c_j, j = 1, ..., M$, where M represents the number of the CoMBs on this path. The proposed algorithm (Algorithm 3) assigns NFs to CoMBs as a sequence of the two following steps:

1. Initially, the algorithm seeks a *first-fit* assignment for each NF, in the order that the NFs have been specified in the request R (lines 2–8). To ensure the correct ordering, for each NF i the algorithm starts searching from the CoMB k (line 3) where the previous NF was assigned. If there are insufficient resources for any of the requested NFs, the request is rejected (lines 7–8).

2. Subsequently, the algorithm performs an *order-preserving worst-fit* for each NF in order to balance the processing load across the CoMBs (lines 9–14). More precisely, the algorithm seeks the worst-fit placement for each NF in the reverse order. This assignment is denoted by a_i. The algorithm essentially restricts the search space for each NF across the CoMBs within the range $\{p_i, ..., a_{i+1}\}$, preserving the correct order of all NFs in the request (i.e., p_i denotes the first-fit assignment for this NF, while a_{i+1} represents the CoMB assigned to the following NF using worst-fit). Finally, the algorithm inserts all NF-to-CoMB assignments into the assignment vector A, which is returned as the solution.

5.4.2 *NFP Assignment*

NFP assignment is carried out based on the NFP utilizations, aiming at load balancing among the NFPs while taking into account any NF location dependencies. The confidentiality of NFP utilizations poses a significant challenge for NFP assignment. One approach is to rely on a third party which collects the NFP utilizations and compares them on behalf of the NFPs. However, information disclosure to a foreign entity raises a lot of concerns and may violate NFP policies. Therefore, we employ another approach which obviates the need for disclosing confidential information to a third party. In particular, we rely on secure Multi-Party Computation (MPC) [57, 45], which allows different parties with private inputs to compute a function on these inputs without revealing their values. MPC essentially executes a cryptographic protocol to perform the computation such that (i) the result of the computation is correct, and (ii) cheating parties will not be able to learn any information about the honest parties inputs.

Algorithm 3 Intra-Provider Middlebox Selection

```
1:  A = ∅, k = 1, p_i = 0  ∀i ∈ {1...N}
2:  for i = 1...N do
3:     for j = k...M do
4:        if d_i ⩽ c_j then
5:           p_i = j, k = j
6:           break
7:        endif
8:     end for
9:     if p_i = 0 then
10:       // processing demand cannot be satisfied, reject request
11:       return ∅
12:    endif
13: end for
14: for i = N...1 do
15:    if i = N then
16:       a_i = argmax_{l∈{p_i...M}} c_l
17:    else
18:       a_i = argmax_{l∈{p_i...a_{i+1}}} c_l
19:    end if
20:    A ⟵ A ∪ a_i
21: end for
22: return A
```

We particularly use secret-sharing MPC [121], where each party splits its private input into shares and distributes them among the other parties. Subsequently, all parties equivalently run an interactive protocol to compute shares of the function output. The parties finally exchange their shares of the output with each other so that the output value can be reconstructed. The privacy-preserving nature of secret sharing as well as the interactive protocol ensures that the input values are not revealed at any point during the protocol execution.

Since we aim at comparing the NFP utilizations, the controllers use the *less than* function provided by the MPC protocol proposed in [45]. This function allows two parties P_1 and P_2 to compare their inputs x_1 and x_2. The main idea behind this protocol is to calculate $c = x_2 - x_1 + 2^l$; if $c > 2^l$, then $x_1 < x_2$ (l denotes the bit-length of x_1 and x_2). Essentially, the outcome of this comparison is designated by the value of the $(l+1)^{th}$ bit of c (i.e., if it is 1, then $x_1 < x_2$). Using the shares of x_1 and x_2 and in several rounds of communication and computation, the protocol computes shares of the $(l+1)^{th}$ bit of c. These shares are combined together to construct the result of the comparison.

5.5 IMPLEMENTATION

In the following, we discuss the implementation of our middlebox, signaling protocols and MPC protocol.

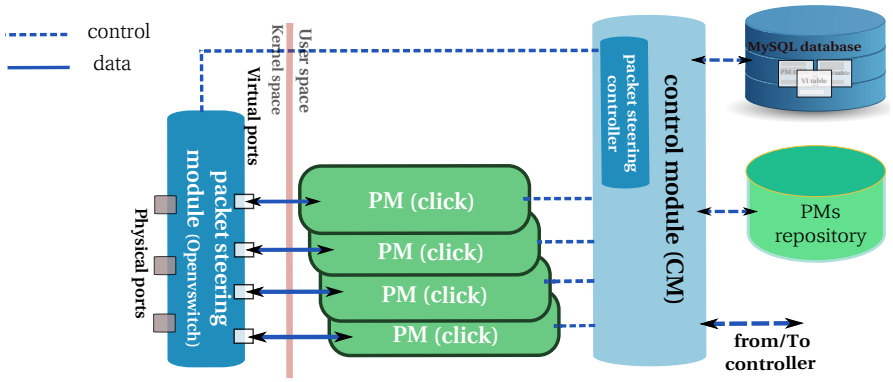


Figure 24: Consolidated middlebox implementation.

5.5.1 *Consolidated Middlebox*

We have implemented a middlebox for the consolidation of multiple NFs on a commodity server. Our CoMB implementation consists of the following components (Figure 24):

Processing Module (PM). A PM implements a specific NF, such as packet filtering, monitoring, or encryption, using Click Modular Router [99] in userspace. Our CoMB supports the instantiation, configuration, and termination of PMs on demand.

Packet Steering Module (PSM). The PSM is responsible for steering the packets of a flow from a physical port to a PM and then back to the CoMB's output port. It can also route a flow between different PMs, when a flow needs to undertake more than one NF that is hosted on the same CoMB. The PSM can identify flows at different granularities by matching packets against pre-defined rules that can contain L2-L4 header fields, as defined by OpenFlow [107]. PSM is implemented with OpenvSwitch [110] running in the Linux kernel. To attach each PM to the PSM, we use *tap* interfaces.

Repository. To speed up the flow processing setup, CoMB uses a repository where PM configuration templates are stored (i.e., Click configuration files). The repository can be synchronized with a central repository managed by the controller to fetch new or updated PM configurations.

Control Module (CM). The CM installs, configures, and terminates PMs according to the instructions it receives from the corresponding controller. To this end, the CM exposes an API to the controller. The CM also configures the rules installed in the PSM for packet steering, using the OpenFlow API. In particular, PM instantiation includes the following steps: (i) creation of the required *tap* interfaces, (ii) retrieving the Click configuration template from the repository, (iii) appending physical/virtual interface names and addresses and any required set of rules to the Click configuration, (iv) installation of the Click configuration, and (v) installation of the flow entries in the OpenvSwitch

flow table for the configuration of packet steering. Upon the PM installation, all configuration information is registered in the CoMB's database.

5.5.2 *Signaling and MPC Protocols*

For the implementation of our signaling protocols, we created the following Click elements:

- *MiddleboxSignaling:* generates the *REQUEST* message sent from the NPCL to the CoMBs along the path.
- *MiddleboxProcessing:* processes incoming *REQUEST* messages on each CoMB and, when needed, updates the controller and CoMB IP address.
- *ControllerSignaling:* encapsulates the *REQUEST* into a *DISCOVERY* message sent from a CoMB to its controller.
- *ControllerProcessing:* processes incoming *DISCOVERY* and *CONTROLLER* messages at each controller.

Middlebox and controller signaling is therefore implemented by combining these elements with existing Click elements for packet I/O and L2/L3 processing. For all our protocols, we encapsulate messages into UDP packets. Furthermore, we use the *Router Alert Option (RAO)* of the IP header to enable CoMBs to distinguish *REQUEST* messages from other packets. MPC is implemented in Python using the Virtual Ideal Functionality Framework (VIFF) [4].

5.6 EVALUATION

In this section, we study the feasibility of MIDAS for on-path processing setup. In particular, we use our Emulab-based experimental facility to measure the processing setup delay and, we further perform simulations to quantify the load balancing level across the NFPs and measure the request acceptance rate.

5.6.1 *Experimental Results*

We run our experiments using 22 commodity servers, each one equipped with an Intel Xeon E5520 quad-core CPU at 2.26 GHz, 6 GB DDR3 RAM and a quad 1G port NICs based on Intel 82571EB. The experiments are conducted with a diverse number of NFPs, each one deploying one controller and three CoMBs in separate servers. In the following, we present the experimental results across 1000 runs.

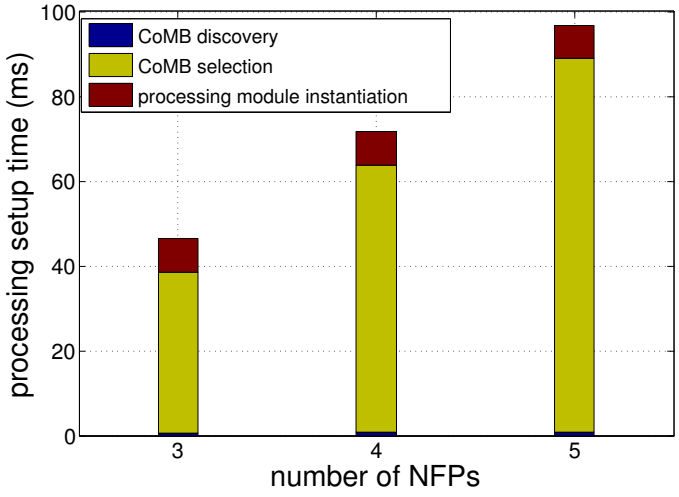


Figure 25: Processing setup delay vs. number of NFPs.

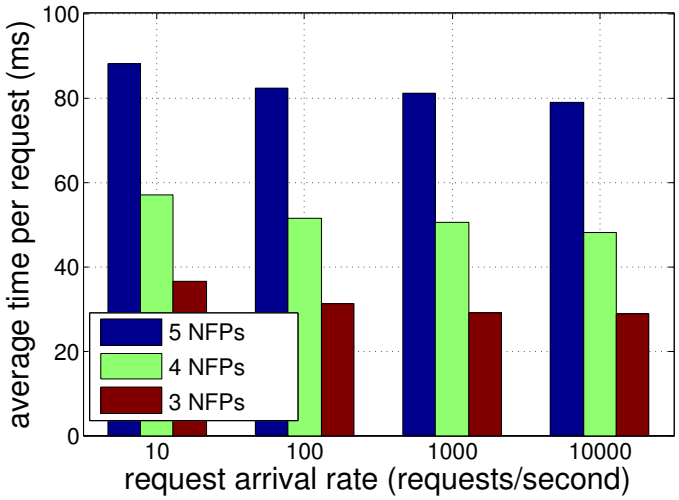
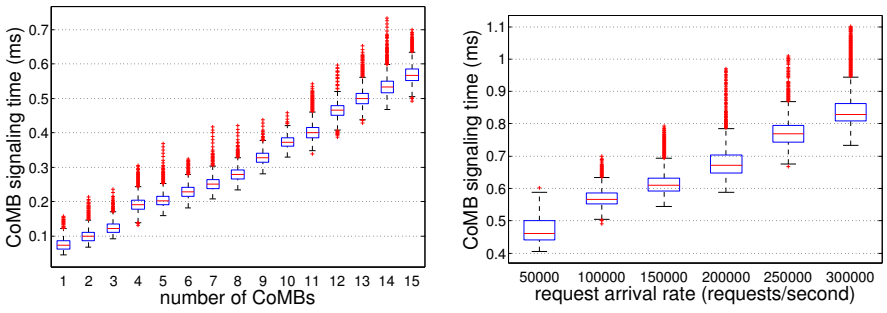


Figure 26: NFP assignment delay (MPC) vs. request arrival rate.

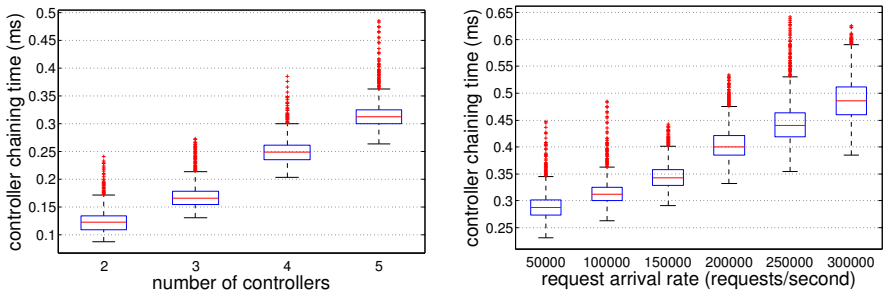
First, we measure the total delay incurred during the flow processing setup. This includes the time elapsed from the generation of a *REQUEST* message until the completion of PM instantiation in the assigned CoMB(s). As shown in Figure 25, the processing setup delay does not exceed 100 ms for up to 5 NFPs, whereas most of the time is spent during the CoMB selection. In contrast, CoMB discovery incurs minimal delay, while PM instantiation takes only a few ms. Note that our measurements mainly show the delays due to packet processing and other computations (i.e., MPC) for on-path processing setup. In a wide-area deployment, these delays will be higher due to the propagation delays along the paths between the end-hosts and between the CoMBs and their controllers.

Figure 26 indicates that the CoMB selection delay stems from MPC during the NFP assignment, while the CoMB assignment within each NFP incurs minimal delay. Although MPC runtime increases with the number of NFPs, the delays with 4 and 5 NFPs, which are more repre-



(a) (b)

Figure 27: CoMB signaling delay vs. a) number of CoMB and b) request arrival rate with 15 CoMBs.



(a) (b)

Figure 28: Controller chaining delay vs. a) number of controllers b) request arrival rate with 5 NFPs.

sentative of the average AS-path length (i.e., 4.2, according to [60]), do not exceed 90 ms. As shown in Figure 26, MPC can handle a diverse number of simultaneous requests without a perceptible impact on the average delay per request. MPC execution can be accelerated by exploiting the latest cryptographic extensions in CPU/GPU instruction sets.

The box-plots in Figures 5.27(a)-5.28(b) provide insights into CoMB signaling and controller chaining. We first measure the delay incurred during middlebox signaling with a diverse number of CoMBs. Figure 5.27(a) depicts that signaling delay is below 0.6 ms for up to 15 CoMBs deployed along the traffic path. To investigate potential implications with a large number of incoming requests, we measure the signaling delay with varying request arrival rates and 15 CoMBs on the path. As shown in Figure 5.27(b), our CoMB can handle 300K *REQUEST* messages per second without a significant impact on signaling delay. Similarly, we measure the controller chaining delay with a diverse number of NFPs and request arrival rates. According to Figure 5.28(a), controller chaining requires less than 0.4 ms for 5 controllers. Controller chaining delay increases in proportion to the request arrival rate but does not exceed 0.6 ms for rates as high as 300K messages per sec with 5 NFPs.

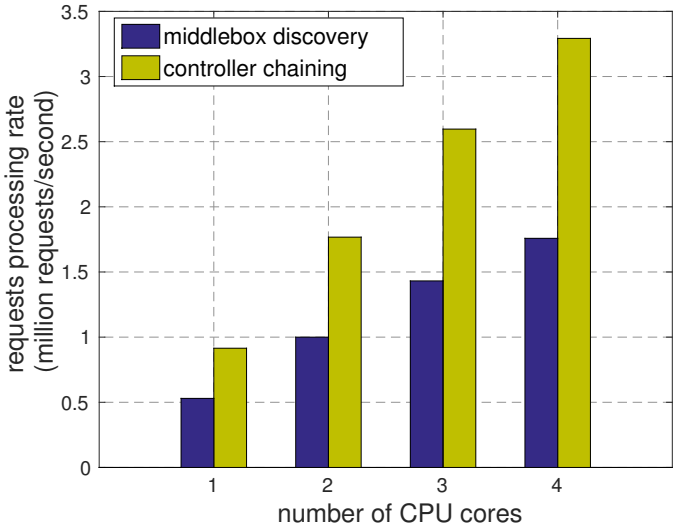


Figure 29: Request processing rate vs. CPU cores.

We further measure the rate that a CoMB and a controller can process messages for middlebox discovery and controller chaining, respectively. Our tests are performed on a single server with optimized packet I/O handling, using a patched Linux kernel (version 2.6.24.7) with the e1000 patched network driver to enable Click packet polling [99]. We parallelize the processing by creating multiple protocol threads, each one assigned to a different CPU core. Figure 29 shows that the processing rate scales linearly up to 4 CPU cores (with shared L3 cache) and reaches millions of messages per second.

	cycles/message	Packet I/O	Protocol operations	Other
Processing at CoMB	4283	48%	25%	27%
Processing at controller	2480	46%	33%	21%

Table 4: Computational requirements for processing setup.

Using Intel VTune Amplifier, we derive the computational requirements (CPU cycles/message) of running our protocols in the CoMB and controllers and further break this down into protocol operations and packet I/O handling. As shown in Table 4, packet I/O accounts for most of the CPU cycles per message, while our protocols do not introduce a large processing overhead. Packet I/O optimizations with Netmap [115], PF_RING DNA [27] or Intel DPDK [1] can further increase the processing rates.

Furthermore, we delve into processing module instantiation with different processing workloads. We specifically measure the delay incurred for PM setup and where the most time is being spent, using the following workloads:

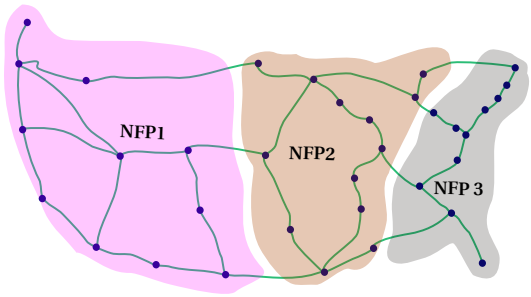


Figure 30: Simulation topology (34 CoMB subdivided into 3 NFPs).

Packet duplication: copies the packets of a flow and sends them through two different ports.

Flow monitoring: collects flow statistics, including data rate, data volume, and flow duration.

Packet filtering: permits or filters flows based on a predefined set of 4000 rules.

Table 5 shows various statistics for PM instantiation. The instantiation of a PM is concluded within approximately 8 ms for all processing workloads. The packet filter yields higher PM installation delay because it requires the installation of the associated rules. However, since PM installation accounts for roughly only 2% of the total PM instantiation delay, the effect of a stateful processing application on the delay is minimal. PM configuration and packet steering require approximately 4 ms each.

	PM config	PM install	Steering	Total
Packet duplication	3.96 ms	<1 µs	3.96 ms	7.93 ms
Flow monitoring	3.97 ms	<1 µs	3.97 ms	7.95 ms
Packet filtering	4.11 ms	87 µs	3.95 ms	8.23 ms

Table 5: PM instantiation time.

5.6.2 *Simulation Results*

Due to the limited scale of our experimental infrastructure, we use simulations to evaluate the level of load balancing with a larger number of CoMBs. We run our simulations using the Internet 2 topology [3], which is subdivided into three NFPs, as shown in Figure 30. Each node in this figure represents the location of a CoMB. Our simulation setup includes 34 CoMBs in total, all of which have the same processing capacity. Processing requests are generated by randomly selecting a pair of nodes and computing the shortest path based on the topology's link costs. The requests include 1 to 3 NFs. The NF(s)

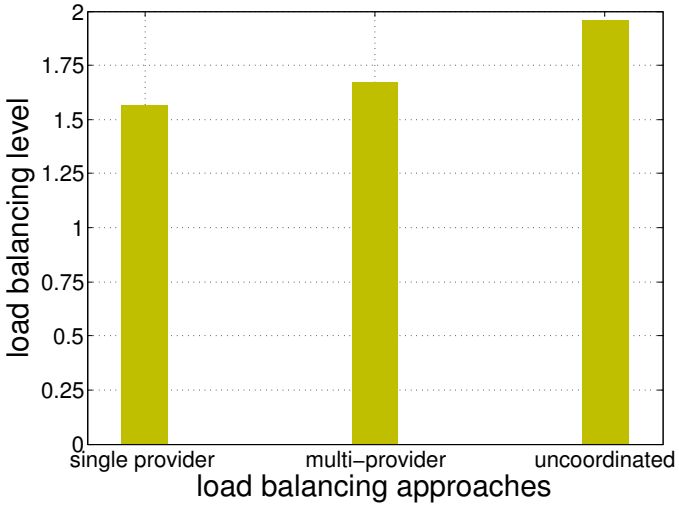


Figure 31: Network-wide load balancing.

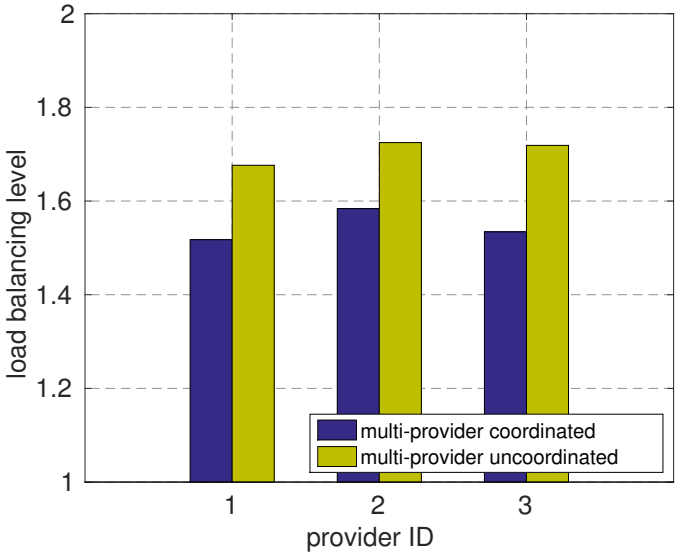


Figure 32: Load balancing within each provider.

and flow rate are randomly sampled from a uniform distribution. All simulation results are obtained across 1000 simulation runs.

We compare the efficiency of MIDAS against two other CoMB selection techniques:

Single-provider. All CoMBs are assigned to a single controller with a network-wide view and complete information about their utilization. The CoMB selection within the NFP is carried out with our algorithm (Section 5.4.1).

Multi-provider uncoordinated. Each CoMB is allowed to select a flow based on a probability that is inversely proportional to its utilization level. This technique does not require any state or coordination among CoMBs and is adapted from [75].

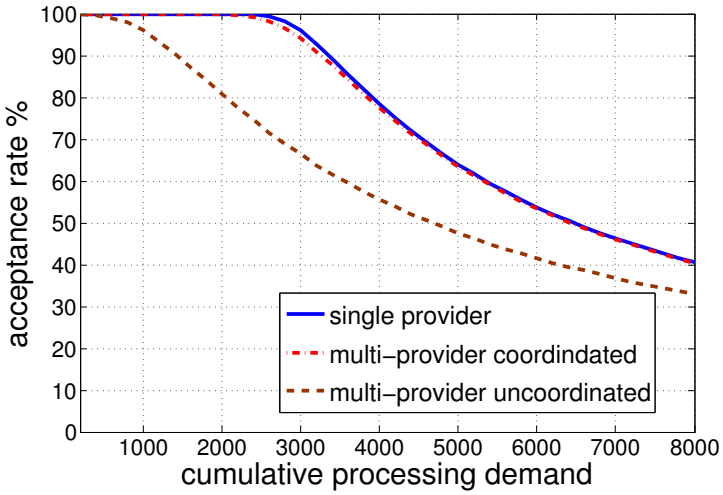


Figure 33: Processing demand acceptance rate.

To perform our comparison, we define the (i) *load balancing level* as the maximum CoMB load over the average CoMB load across the NFPs and (ii) *processing demand acceptance rate* as the proportion of the *processing demand* being accepted. Lower values of the load balancing level represent better load balancing, whereas a value of 1 designates optimal load balancing. Processing demand acceptance rate is an efficiency metric from the perspective of an NFP, as a high acceptance rate can generate more revenue.

Figure 31 illustrates the load balancing level across the network with each of the three techniques. Our CoMB selection approach (denoted as "multi-provider coordinated") achieves an adequate level of load balancing and only slightly inferior to the "single-provider" technique, where the controller has a global network view. The uncoordinated approach yields load imbalance, which indicates the significance of coordination for CoMB selection. Furthermore, Figure 32 depicts the load balancing level achieved within each NFP with our technique and the uncoordinated approach (i.e., the "single-provider" technique aims at network-wide load balancing and, therefore, is not directly comparable). The balanced load within each provider's network validates the efficiency of MPC for intra-provider middlebox selection. According to Figure 33, MIDAS and "single-provider" approach are able to maintain a high acceptance rate as the processing demand increases. On the other hand, the "multi-provider uncoordinated" approach exhibits an inefficiency in accommodating processing requests and eventually leads to a large number of rejections.

5.7 RELATED WORK

In this section, we discuss the related work on signaling protocols, flow processing architectures and middlebox management.

Signalling Protocols: Protocols for middlebox signaling have been primarily designed for device configuration and control and are usually specific to a certain middlebox function. In particular, signaling protocols such as SIMCO [126], NAT/firewall NSIS [127], STUN [116] and NUTSS [83] enable applications to dynamically configure NAT and firewall devices deployed along the traffic path such that the application's flow can traverse the middleboxes. On the other hand, QoS signaling protocols such as RSVP [47] and QoS NSIS [87, 105] are designed to fulfil QoS demands by reserving network resources and maintaining the reservation state on routers and platforms located on the traffic path. QoS signaling protocols can be extended to support resource reservation (i.e., reserving computing and network resources) for NFs deployment. However, due to their distributed nature (similar to [75]) and the correctness requirement of service chains, these protocols can only support first-fit resource allocation leading to load imbalance and resulting in inefficient resource utilization. In contrast, we propose signaling protocols which enable centralized resource discovery and allocation (through the NFP's controller) within each provider's network and require no state maintenance per platform leading to a network-wide visibility, efficient resource allocation (i.e., load balancing) and simpler platform design.

Flow Processing Architectures. Authors in [122] propose an architecture, namely APLOMB, for the outsourcing of flow processing to the cloud. APLOMB relies on widely used techniques for traffic redirection such as tunneling and DNS. Flowstream [81] outlines an architecture for distributed flow processing by coupling commodity servers with programmable switching hardware such as OpenFlow. Both APLOMB and Flowstream provide support for flow processing off the traffic path. An architecture for the customization of the network is presented in [132]. This architecture encompasses service nodes where NFs are deployed and a service controller per provider which is responsible for resource management and connection setup. This architecture has similar components to MIDAS, and both are designed for on-path processing. However, we additionally propose signaling protocols and a middlebox selection algorithm. We further provide a prototype implementation and an experimental evaluation of MIDAS.

Middlebox Management. Recent work presents an architecture for the resource management of software middleboxes, where a centralized controller assigns flows to middleboxes in order to achieve load balancing [119]. This architecture aims at facilitating the management of middleboxes in enterprise networks. In contrast, we propose solutions for the discovery and selection of middleboxes along the traffic path. Furthermore, authors in [89] investigate techniques for the distribution of IDS load across multiple nodes and the assignment of flows to these nodes using a centralized controller with a global net-

work view. This is complementary to our work and can be used to further improve the level of load balancing with MIDAS by splitting expensive processing tasks into multiple CoMBs.

5.8 SUMMARY

In this chapter, we have presented an architecture for the coordination of on-path processing setup to circumvent the difficulty in CoMB discovery and selection across multiple NFPs, without any prior knowledge of the network path. We have proposed signaling protocols for CoMB discovery and NFP interoperability and presented an algorithm for order-preserving CoMB selection within each NFP. For NFP assignment, we have leveraged on MPC to preserve the confidentiality of CoMB utilizations across NFPs.

Using our prototype implementation, we have shown that MIDAS incurs processing setup delays in the order of tenths of milliseconds. In particular, CoMB discovery and controller chaining incur low delays with a large number of CoMBs and high request arrival rates. Our experimental results show that MPC-based NFP assignment accounts for most of the processing setup delay. Nevertheless, MPC does not introduce a scalability limitation because the MPC delay (i) is not affected by the request arrival rate and (ii) does not significantly increase with a number of NFPs that does not exceed the average AS-length of Internet paths. Coupling MPC with our middlebox selection algorithm results in network-wide load balancing and high request acceptance rates, as shown in our simulation results. We believe that our work indicates the feasibility of on-path processing and takes a step towards empowering the *middle* and bringing more flexibility and intelligence to the network.

6

DISTNSE: DISTRIBUTED NETWORK SERVICE EMBEDDING FOR OFF-PATH FLOW PROCESSING

In chapter 5, we proposed an architecture to deploy NFs on the traffic path. On-path processing obviates the need for traffic redirection which can lead to latency inflation and high bandwidth consumption. However, on-path processing also requires the presence of processing platforms with sufficient available resources along the traffic path and might force all the traffic to pass through the on-path middleboxes, resulting in unwanted processing on the traffic and introducing a single point of failure. On the other hand, off-path processing provides plenty of resources by redirecting traffic to DCs (or micro-DCs) deployed off the traffic path, avoiding unwanted processing and possible disruption due to middlebox failure.

However, akin to on-path processing, outsourcing NFs to DCs deployed off the traffic path should preserve the order and comply with the location constraints of NFs. This, in turn, raises significant challenges in terms of Network Service Embedding (NSE), which aims at assigning NFs to network paths and DCs. In particular, due to the limited geographical footprint, a single NFP may not fulfil the location-dependencies of all NFs in a service chain leading to the need for multi-providers NSE. However, as discussed in the introduction, NFPs have been traditionally known for their secrecy about their network resources. They further tend to autonomously implement different local policies (e.g., load balancing). Consequently, an efficient off-path NSE should aim at maintaining the NFP privacy and autonomy, while accounting for the requirements of NF deployment (i.e., correctness and location-dependencies). Furthermore, as with any service offering, NSE should enable competitive price offering to the client.

To this end, in this chapter we propose DistNSE, a distributed architecture that enables the collaboration between NFPs to perform NSE[1], while maintaining the privacy and the autonomy of the participating NFPs. DistNSE ensures competitive pricing by enabling different providers to compete for different NFs of a service chain based on their own policies. DistNSE decomposes embedding in two steps: inter-provider embedding and intra-provider embedding. Inter-provider embedding is carried out using a distributed algorithm to partition a service chain into multiple subchains for which different NFPs can compete. On the other hand, intra-provider embedding maps subchains to network paths and DCs, while complying

1 Unless mentioned otherwise, in this chapter we refer to off-path NSE as NSE.

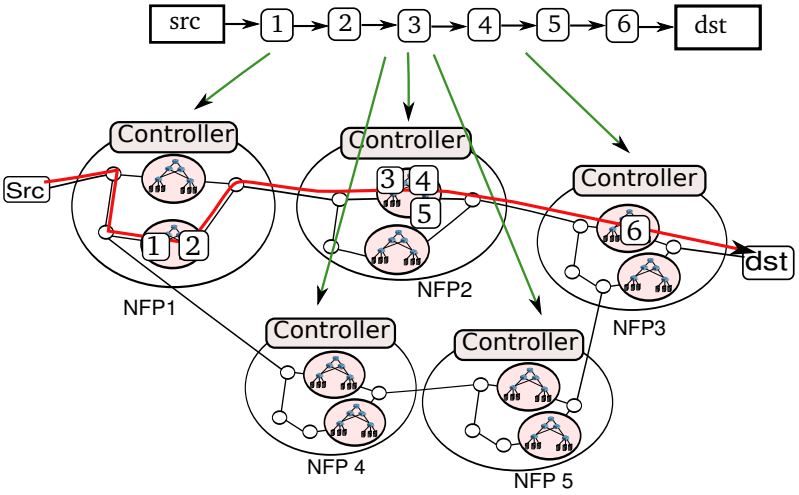


Figure 34: Assignment of NFs to NFPs.

with two different policies: (i) service cost minimization, where the NFP aims at minimizing embedding cost while ensuring minimum competitive service price and (ii) load balancing, where the NFP distributes the traffic load across the network to improve reliability and robustness. Furthermore, to exchange information about the generated subchains and the competing NFPs, DistNSE augments the inter-provider algorithm with a communication protocol. We use simulations to assess the efficiency of DistNSE and identify significant gains over Polyvine [54], a widely-known distributed embedding framework.

The remainder of this chapter is organized as follows. In Section 6.1, we discuss the challenges and requirements of NSE. Section 6.2 introduces our network model. In Section 6.3, we present our network service embedding algorithms, while in Section 6.4 we describe our communication protocol. In Section 6.5, we evaluate the efficiency of DistNSE and discuss our simulation results. Section 6.6 provides an overview of related work. Finally, in Section 6.7 we highlight our conclusions.

6.1 CHALLENGES AND REQUIREMENTS

In this section, we discuss the challenges and the requirements of NSE across multiple providers.

To express network policy and requirements, NFs are represented through service chains. In line with NFaaS model, we envision clients outsourcing service chains from the enterprise network to NFPs' networks. Accordingly, we consider NFPs offering processing and BW capacity to host service chains by deploying micro-datacenters in their network (Figure 34). We further assume the presence of a central controller in each NFP network. This controller has full knowledge of

and full access to the network and processing resources offered by the NFP.

Figure 34 shows an example of embedding a service chain by NFPs. In particular, 6 NFs have to be assigned to the DC's servers and links within selected NFPs (NFP 1, 2 and 3). Furthermore, a path has to be established between the source and destination to carry the service chain traffic. To select DCs' servers and links as well as identify a network path for a service chain, we need to fulfill a set of requirements:

- **Correctness:** The order of the service chain should not be violated, i.e., NFs should be assigned to network and DCs such that the traffic traverses them in the right order. This should be maintained within and across NFPs (Figure 34).
- **Privacy:** NFPs are usually reluctant to reveal any information about their network such as topology, link utilization and servers' utilization. For instance, ISPs advertise their topologies using simplified graphs [124] with no information about router-level connectivity or PoPs. Cloud providers such as Amazon [10] announce their service price using a type-based model where a type is a combination of resources with an associated price. As a result, NSE should enable the collaboration between providers without violating the privacy of each provider.
- **Autonomy:** NFPs usually implement different policies to run their networks. For instance, some NFPs might aim at lowering service price to attract more clients and increase revenue, others might target attaining load balancing to increase reliability. NSE should enable NFPs to implement different policies independently of each other.
- **Competition:** As with any service offered today, a service client expects different offers for embedding a service chain. Hence, NSE should allow NFPs to compete for embedding a service chain and to provide different service prices to the client.

Fulfilling these requirements raises several challenges for multi-provider NSE. These challenges stem from the confidentiality as well as the autonomy each NFP maintains. In other words, each NFP aims to autonomously implement its own embedding policy while being reluctant to share information about its policy and resources utilization with other NFPs. Subsequently, global and complete knowledge about all NFPs' networks cannot be assumed when performing NSE.

A recent work [62] enables multi-provider NSE by abstracting the NFP network to a DC-level topology, where nodes represent NFP DCs with their service cost (as a node attribute) and links represent inter-DC connectivity with weights proportional to the DCs utilization. This work further proposes a central coordinator to collect the

advertised topology and perform NSE. While this approach restricts the amount of disclosed information, it still reveals information about the provider's policy in terms of pricing and DCs utilization to a third party. Furthermore, relying on a third party for assigning NFs to NFPs may raise concerns about reliability (i.e., a single point of failure), market fairness (i.e., an NFP is favoured over other NFPs to host a service request) and confidentiality (i.e., the coordinator sharing information with the competing NFPs).

Another work is Polyvine [54], a framework for partitioning and assigning a virtual network across multiple providers. It enables each provider to select a segment of the virtual network and forward the rest to the next providers. Polyvine can be considered for distributed NSE. However, we need to account for the difference between virtual networks and service chains. Virtual networks do not assume a particular order in contrast to service chains, i.e., virtual networks are usually undirected graphs. Furthermore, with Polyvine different NFPs cannot compete for the same segment of the virtual network. Specifically, as soon as an NFP selects a virtual network segment to embed, other NFPs on the AS-path cannot compete for that segment.

To meet the requirements of NSE, we propose two-steps distributed NSE approach: (i) inter-provider embedding, where we develop a distributed algorithm allowing different NFPs to compete for embedding a service chain while preserving NFPs privacy and autonomy, and (ii) intra-provider embedding, where we present algorithms for DC and network path selection as well as NFs to DC mapping while considering different policies. We further augment our architecture with a communication protocol to coordinate distributed NSE.

6.2 NETWORK MODEL

In this section, we present the models of our NFs service chain and NFPs network.

6.2.1 *Service Chain Model*

We represent a service chain as a directed graph $SC = (N, L)$, where N is the set of NFs, and L is the set of links between nodes of set N. Each NF i has computation demand which is denoted with d_i. Each link $l \in L$ is associated with a bandwidth demand B (i.e., B has the same value for all links of the service chain).

6.2.2 *Network Model*

We consider a multi-provider network where each NFP network is an administratively-independent domain comprising of a set of interconnected Micro-DCs. Micro-DCs are small datacenters with a wide

geographical footprint (deployed at point of presence and network aggregation points). A micro-DC typically consists of one or more racks with 40 servers per rack, and two routers and two switches connected in a mesh to provide redundancy and load-balancing [78]. We model our network in three layers as follows:

NFP-level network: We represent the inter-providers network as a weighted undirected graph $G_{as} = (A_{as}, L_{as})$, where A_{as} is the set of NFPs, and L_{as} is the set of peering links between the NFPs of the set A_{as}.

NFP substrate network: We rely on an undirected graph $G_S = (V_S, E_S)$ for the description of NFP network topology. V_S is the set of nodes, and E_S is the set of links between nodes of set V_S. Nodes consist of routers R_S, microdatacenters D_S and peering nodes P_S such that $V_S = R_S \cup D_S \cup P_S$. Each graph edge $(u, v) \in E_S$ between node u and v is associated with a weight w_{uv} (assigned by the NFP) and a residual capacity r_{uv}.

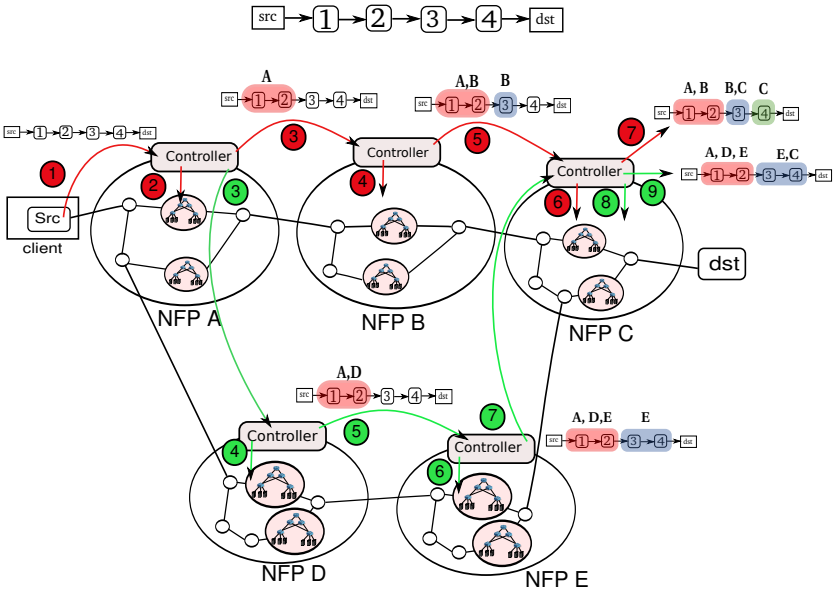
Micro-DC network: We consider a hierarchical two-level multi-rack micro-DC network. We represent the DC topology using an undirected graph $G_{dc} = (V_{dc}, F_{dc})$. V_{dc} denotes the set of servers and switches, and F_{dc} represents the set of links connecting the servers and switches. A DC consists of a set of racks R_{dc}. Each rack q has a set of servers S^q and aggregation links F^q connecting the rack to the DC root switches. Each server $s^q \in S^q$ is associated with residual processing capacity $cp(s^q)$ and an access link f_s^q with residual capacity $rc(f_s^q)$. Each aggregation link $f^q \in F^q$ has residual capacity $rc(f^q)$.

6.3 NETWORK SERVICE EMBEDDING

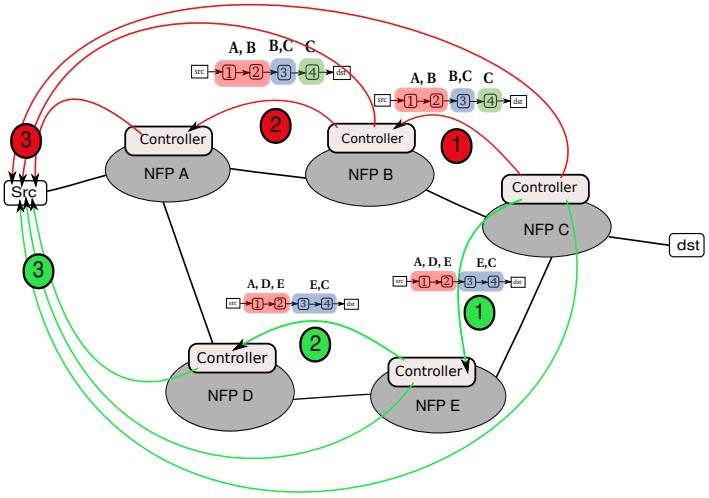
In this section, we present our embedding algorithms. We start with an overview of the embedding workflow and proceed with a detailed description of our two-steps NSE: inter-provider embedding and intra-provider embedding.

6.3.1 *Embedding Overview*

Figure 35 shows an overview of our distributed NSE approach. NSE starts with the client submitting its service chain request (Figure 6.35(a) step 1) to its home NFP (a home NFP can be the client's access ISP or an NFP trusted by the client). Subsequently, the home NFP identifies a possible embedding (Figure 6.35(a) step 2) for a subchain (e.g., NF 1 and 2 are a subchain) of the service chain and accordingly announces itself as a candidate for the generated subchain. This is followed by forwarding a copy of the subchain and the service chain to each of the downstream NFPs (to NFP B and D in Figure 6.35(a) step 3). As in NFP A, NFP B and D identify a possible embedding to generate a new subchain and/or deploy already-generated subchains. This is



(a) Chain partitioning and competition establishment.



(b) Bidding for the service chain.

Figure 35: Overview of distNSE workflow.

followed by forwarding the subchains and the service chain to other downstream NFPs for further embedding. The forwarding continues until reaching the home NFP of the destination (NFP C). After performing local embedding, NFP C sends the final set of subchains in the reverse direction through the NFPs toward the source (Figure 6.35(b) step 1 and 2). Subsequently, each NFP submits a bid (Figure 6.35(b) step 3) to embed the corresponding subchains (the subchains it identified earlier for embedding).

6.3.2 *Inter-Provider Embedding*

Inter-provider embedding aims to enable the collaboration between NFPs to embed a service chain such that competitive service pricing, the correctness of the service chain, and the privacy and the autonomy of each NFP are ensured. A straight forward approach is to allow NFPs on each AS-path to the destination (see Figure 6.35(a)) to identify all possible subchains of a service chain such that the NFs' order and continuity within each subchain are preserved (also known as finding all possible substrings on a string [129]). Subsequently, the NFPs combine the generated subchains to identify all possible mappings of subchains to NFPs such that: (i) the NFs' order across subchains mapped to different NFPs is maintained, (ii) an NF should not belong to two different subchains of the same mapping, i.e., for every subchain sc_k in a mapping consisting of m subchains, $\cap_k^m sc_k = \emptyset$, and (iii) the union of all subchains within a mapping should equal the original service chain, i.e., $SC = \bigcup_{k=1}^{m} sc_k$. For each correct mapping, each NFP announces its service cost such that the mapping with the minimum cost is identified.

While providing the optimal solution for each path, this approach has the time complexity of $O(2^{|N|-1})$, where $|N|$ denotes the number of NFs in a service chain. This, in turn, requires an exponentially growing number of messages to exchange information about the possible mappings. Furthermore, since each NFP has to reveal the service price for each possible mapping of subchains to NFPs, competitive and fair pricing cannot be ensured, i.e., NFPs may adapt their service costs based on the prices revealed by their competitors.

To this end, we propose a heuristic distributed algorithm which partitions a service chain into a set of subchains for which different NFPs can compete. Our algorithm works in a per AS-path fashion, i.e., for each AS-path a different algorithm instance is launched to partition the service chain (red and green arrows in Figure 6.35(a)). The result of each instance is reported to the client such that she can choose the path with the minimum service cost. For each path, our algorithm has the time complexity of $O(p)$, where p is the number of NFPs on an AS-path. The algorithm works in two phases:

1. **Service chain partitioning and competition establishment:** Figure 36 shows an example of the first phase of the algorithm. This phase starts when the first provider on an AS-path (NFP A) receives a service chain request SC. The NFP generates the first subchain sc_1 out of the SC. The number of NFs on the subchain depends on the NFP policy and the available resources. Upon generating the subchain, the NFP (i.e., NFP A) nominates itself as a candidate to host sc_1. We define the mapping of NFPs to a subchains set SUB as $M_{SC} : SUB \longleftarrow A_{as}^{SC}$, where A_{as}^{SC} denotes the set of NFPs willing to compete for SC. The NFP A passes the generated SUB with the SC to the other providers downstream. The next NFP (i.e., NFP B) starts by embedding the NFs which have not yet been embedded leading to the generation of a new subchain (sc_2) with the NFP B as a candidate for it. Then, NFP B tries to embed the existing subchains (i.e., sc_1) and adds itself as another embedding candidate. Next, NFP B forwards the generated subchains and the original chain to the next NFP on the path where the same embedding steps are performed. The forwarding continues until reaching the last NFP which concludes the first phase of the algorithm. In order to avoid violating the service chain order, an NFP can only compete for the last generated subchain. For example, NFP C can only compete for sc_2 out of sc_1 and sc_2 because if it competes for both and wins sc_1, whereas NFP B competes for and wins sc_2, the service chain order is violated.

2. **NFP selection:** This phase aims to select the set of NFPs which incurs the minimum service cost. It starts when the last NFP on the AS-path sends the final mapping of subchains in the reverse direction through the NFPs (step 1 and 2 in Figure 6.35(b)). We consider two different approaches for NFP selection. The first one includes the NFPs submitting their bids to the client which subsequently selects the set of NFPs with the minimum service cost. The second approach is to use Multi-Party Computation (MPC) [45] [63] to enable secure bidding between the NFPs for the different subchains. The first approach does not require further processing by the NFPs; however, it reveals the bids of the NFPs to the client. On other hand, while the MPC approach hides the NFPs bids and only reveals the winning NFPs, it is computationally-intensive and requires more communication rounds between the NFPs. We study the MPC approach in Chapter 5. Hence, in this chapter we focus on the first approach.

With over 45,000 ASes on Internet today [60] [9], forwarding a service chain (Figure 6.35(a)) to every possible NFP (if each AS is an NFP) is deemed impractical and might significantly prolong the con-

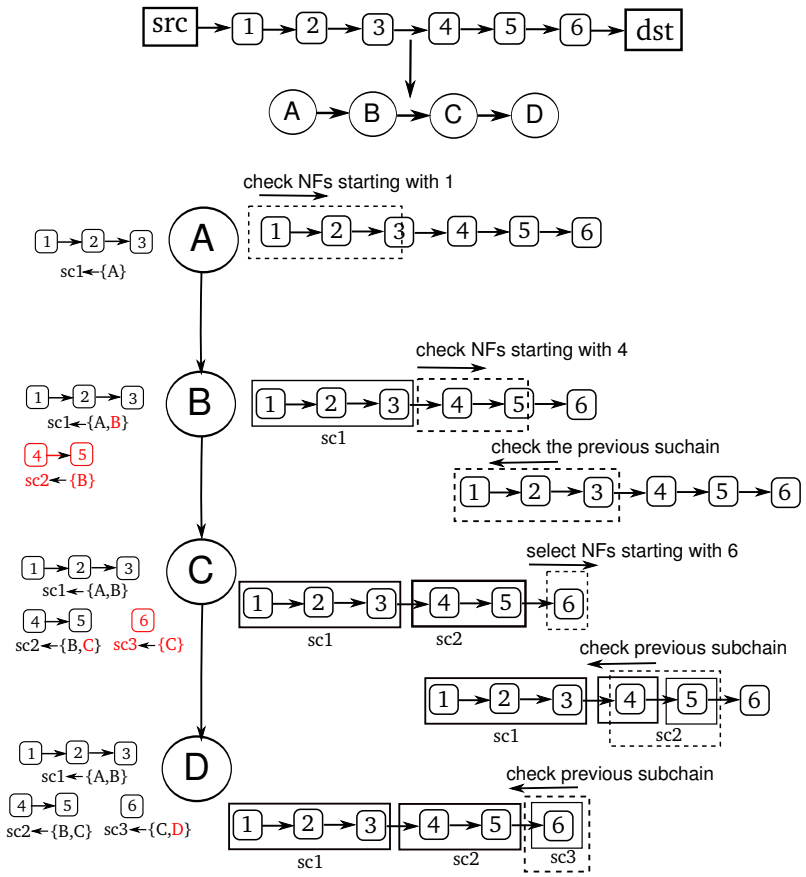


Figure 36: Example of inter-provider service embedding.

vergence time of NSE. To restrict the search space for a service chain, we discuss two methods:

1. **BGP router queries.** The client's home NFP utilizes information from BGP routers to select an AS-path or a set of paths along which NSE is performed. In particular, similar to an ISP, a NFP identifies paths which comply with its policy (*e.g.*, paths that do not incur peering costs).

2. **Time to live:** Since the advertised BGP paths usually depend on the view and the policy of the neighbouring NFPs, we propose an alternative approach to restrict the search space. In particular, we let the client specify a threshold on the number of NFPs a service chain can be forwarded to. The threshold can be calculated based on the BGP routing table or statistics collected to measure the AS-path length [60] [9]. The threshold can be encoded as a TTL value in the message carrying the service chain (see Section 6.4). Accordingly, it is decremented by each NFP receiving the service chain request, and upon reaching zero, the request message is discarded. This approach does not require maintaining

any path state by the NFPs, and it extends the forwarding space beyond the BGP routers' view of the neighbouring NFPs. However, finding a suitable threshold might be challenging. A recent study [60] found that the average number of ASes per path has stayed almost constant at about 4.2 for the last 12 years. Such information can be helpful to adjust the threshold.

6.3.3 *Intra-Provider Embedding*

Intra-provider embedding aims at mapping service chain NFs to servers within a DC and connecting them with the other NFP networks (through peering nodes) using DC and network links. Traditionally, NFPs have been implementing different network management policies depending on the their business model, network structure and available resources. Accordingly, we propose an intra-provider embedding algorithm considering the following objectives:

- **Minimizing service cost:** To attract more clients and compete with other NFPs, an NFP implements a policy aiming at mapping NFs to servers and links such that minimum service cost is incurred. In particular, the NFP seeks embedding the largest possible number of NFs (a subchain) of a service chain such that the lowest possible amount of resources is consumed. To facilitate such embedding, we define *embedding efficiency ratio (EER)* as a metric to measure the efficiency of a particular embedding for the NFP. In particular, EER_k for a subchain $sc_k = (N_k, L_k)$ represents the *total embedding resources demand for* sc_k*/total resources demand for* sc_k. EER specifies the gap between the physical resources an NFP has to provide to embed sc_k and the sc_k actual demand. Hence, the lower the value of EER_k, the lower the cost of embedding in terms of resources consumed, a value of 1 for EER_k represents the optimal embedding for sc_k. We formally define EER_k as:

$$EER_k = \frac{\sum_i d_i^k + B^k \, [|F_{dc}^k| + |E_s^k|]}{\sum_i d_i^k + B^k \, |N^k|} \qquad (2)$$
$$\forall i \in N^k, F_{dc}^k \in F_{dc}, E_s^k \in E_S$$

The denominator accumulates the CPU and BW demand of a subchain sc_k, whereas the nominator accumulates the CPU and BW resources required to embed sc_k. F_{dc}^k and E_S^k represent the sets of DC links and network links to which sc_k can be assigned, respectively. We further define the service cost (CR_k) for a subchain sc_k as:

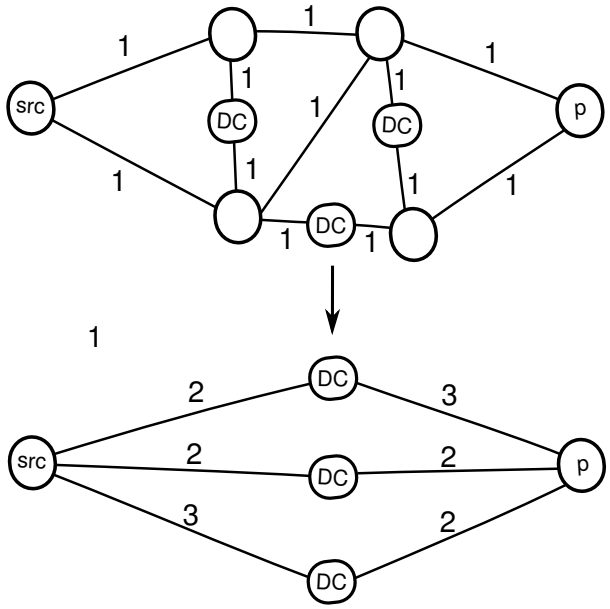


Figure 37: An example for NFP DC topology conversion with the numbers representing the weight of each link.

$$CR_k = C_{CPU} \sum_i d_i^k + C_{BW}^{dc} B^k |F_{dc}^k| + C_{BW}^S B^k |E_s^k| \quad (3)$$
$$\forall i \in N^k, F_{dc}^k \in F_{dc}, E_s^k \in E_S$$

The service cost is the accumulation of CPU and BW embedding resources multiplied by the CPU cost per unit (C_{CPU}) and BW cost per unit in DCs (C_{BW}^{dc}) and in substrate links (C_{BW}^S).

- **Load balancing:** Load balancing is a popular policy in ISP networks. To implement load balancing, we alter the links' weight w_{uv} as a function of the links' load using a simple formula: $w_{uv} = w_{uv}^0 + \beta\ load_{u,v}$, where w_{uv}^0 denotes the initial weight of link (u, v) and β provides a factor to adjust the effect of the link load on the link weight. Accordingly, we use Dijkstra's algorithm to map a subchain to links with minimum load.

Considering these policies, we develop an intra-provider embedding algorithm which is called by the NFP when receiving a service chain SC_t or a set of subchains SUB. For each subchain or service chain, the algorithm aims to embed as many NFs as possible. In particular, it seeks to assign NFs to links and servers within a DC and select a network path traversing the DC between the traffic source src (a peering node or a client) and one of the peering nodes ($P_S^t \in V_S$) connecting the NFP to a downstream NFP. If an embedding does not assign all of the NFs of a subchain, a new subchain with the assigned NFs is generated. The output of this algorithm is fed as an input to

the inter-provider algorithm (see Figure 35 and Section 6.3.2). The algorithm consists of two stages:

1. **Network path and DC selection:** In this stage (Algorithm 4), the algorithm selects a network path and a DC between the traffic source src and a peering node $p \in P_S^t$. In particular, for each pair of src and p, we use the CONV_TOPO() function (based on the algorithm in [53]) to convert the NFP topology to a smaller topology with DCs and the pair of src and p. Each link in the new topology which connects a DC to either src or p has a weight corresponding to the total weight of the shortest path on the original topology (see the example in Figure 37). Next, in the new topology, we calculate and sort all the paths (PATHS) between src and p in a increasing order based on the total weight per path. Then, by going through the sorted paths, we identify the path incurring the minimum EER per path (EER_{path}). This includes performing embedding within each DC (using the DC_MAPPING() function) and calculating the EER_{path} for the DC and the network links. When calculating the EER_{path}, we map the path in the new topology back to original path in the NFP topology. We further compare the minimum EER_{path} across different pairs of src and p to identify the pair of src and p with the minimum EER. The final output is a mapping M of NFs to DC servers and network links. This algorithm implements directly the minimizing service cost policy by calculating the total weight of the links in the new topology as the number of links on the original topology. This leads to the minimization of the number of links, hence the minimization of service cost (eq. 3). To implement load balancing, we simply update the weights as a function of the load as discussed earlier. This algorithm has the time complexity of $O(|P_S||D_S|)$, where $|P_S|$ is the number of peering nodes and $|D_S|$ is the number of DCs in the NFP's network.

2. **DC mapping:** Embedding a service chain in a DC with capacity constraints can be formulated as a multi-commodity flow problem which is an NP-hard problem. Therefore, we propose a heuristic algorithm (Algorithm 5). Our algorithm generates an embedding of the full chain or a subchain sc_k depending on the available capacity on the DC. The algorithm starts with sorting the racks R_{dc} in decreasing order based on the residual capacity $rc(f^q)$ of the aggregation link of each rack q. Subsequently, the algorithm goes through the racks where it again sorts the servers S^q in each rack in decreasing order based on the processing residual capacity $cp(s^q)$ of each server s^q. To each server s^q, the algorithm tries to assign as many NFs as possible. The algorithm aims at using the lowest possible number of servers and

links to reduce the embedding cost. The output of this algorithm is the mapping M_{dc} of NFs to servers and service chain bandwidth B^k to links. The time complexity of this algorithm is $O(|R_{dc}||S^q| + |N|)$ where $|R_{dc}|$ is the number of racks, $|S^q|$ is the number of servers per rack and $|N|$ is the number of NFs in a service chain.

Algorithm 4 Path and DC Selection

Inputs: $G_S = (N_S, L_S), sc_k, src$

$EER \leftarrow \infty$
$M \leftarrow \emptyset$
for each $p \in P_S^t$ **do**
  $EER_{path} \leftarrow \infty$
  $M_{path} \leftarrow \emptyset$
  $G_s^{new} = CONV\_TOPO(G_S, src, p)$
  $PATHS = FIND\_ALL\_PATHS(G_S^{new})$
  SORT_INC(PATHS) // sort paths based on total weight
  for each $path \in PATHS$ **do**
    M_{dc} =DC_MAPPING($d \in path, sc_k$)
    $eer = CAL\_EER(path, M_{dc})$
    if $eer < EER_{path}$ **then**
      $EER_{path} = eer$
      $M_{path} = (path, M_{dc})$
    end if
  end for
  if $EER_{path} < EER$ **then**
    $EER = EER_{path}$
    $M = M_{path}$
  end if
end for
return M

6.4 DISTNSE PROTOCOL

To exchange the embedding information, we design a communication protocol with six messages sent and received asynchronously among the NFPs and with the client as shown in Figure 38:

1. REQUEST($req\_id, ttl, SC, M_{SC}, src, dst, path$): This message is generated by the client to initiate NSE embedding between a traffic source src and a destination dst. The REQUEST message also communicates the mapping of providers to subchains (M_{SC}) between the NFPs (see Section 6.3.2). As discussed in Section 6.3.2, the REQUEST message also carries a TTL field, set by the client's home NFP, to restrict the number of NFPs a REQUEST message can be forwarded to. Furthermore, the $path$

Algorithm 5 DC Mapping

Inputs: $G_{dc} = (S_{dc}, F_{dc}), sc_k = (N_k, L_k)$

$M_{dc} = \emptyset$
$M_{servers} = \emptyset$
SORT_DEC(R_{dc}) // sort racks based on aggregation links capacity
for each $q \in R_{dc}$ **do**
  if $B^k > rc(f^q)$ **then**
    return $\emptyset$
  else
    SORT_DEC(S^q) // sort servers based on capacity
    $m = \emptyset$
    for each $s^q \in S^q$ **do**
      for each $i \in N_k$
        if $d_i <= cp(s^q)$ and $B^k <= rc(f_s)$ **then**
          $M_{servers} = M_{servers} \cup (i \rightarrow s^q)$
          $N_k = N_k \triangle \{i\}$ // remove n from NFs
          if $N_k = \emptyset$ **then**
            break
          end if
        else
          break
        end if
      end for
    end for
  end if
  if $M_{servers} \neq \emptyset$ **then**
    $M_{dc} = M_{dc} \cup (M_{servers}, f^q)$
    if $N_k = \emptyset$ **then**
      break
    end if
  end if
end for
return M_{dc}

field collects the IDs of the NFPs traversed by the REQUEST message such that the combination of the $path$ and req_id fields identifies the embedding performed per path for a particular request.

2. REVERSE($req\_id, SC, M_{SC}, path$): In response to the reception of a REQUEST message, a REVERSE message is generated by the home NFP of the destination (NFP D in Figure 38) to convey the final mapping M_{SC} to the upstream NFPs on the path $path$.

3. OFFER($req\_id, sub_{nfp}, M_{SC}, bid, path$): Upon receiving the REVERSE message, each NFP in the M_{SC} (and on the path $path$) sends a bidding price (carried in the bid field) to compete for the set of subchains sub_{nfp} (it selected earlier) using an OFFER message.

4. EMBED($req\_id, sub_{nfp}, path$): Based on the bids received through the OFFER messages, the client sends an EMBED message to the winning NFPs. The EMBED message carries the set of subchains sub_{nfp} an NFP should embed.

5. SUCCESS($req\_id, sub_{nfp}, path$): This message is sent by the NFP in case of successful embedding of the set of subchains sub_{nfp}.

6. FAILURE($req\_id, sub_{nfp}, path$): This message is sent by an NFP in case of failed embedding or a zero ttl value of a REQUEST message. A FAILURE message responding to zero ttl carries an empty set of subchains sub_{nfp}. This message is also sent by the the home NFP of the destination, instead of a REVERSE message, if the mapping (i.e., M_{SC}) in a REQUEST message does not contain all the NFs of the service chain (i.e., incomplete embedding).

6.5 EVALUATION

To evaluate the efficiency of DistNSE for NSE, we implemented a simulator in Python. The simulator generates the various protocol messages and handles resource search and allocation within each NFP. We ran our tests on a server with four-core Intel Xeon CPUs at 2.26 GHz and 6 GB RAM.

We use IGEN [2] to generate AS and intra-provider topologies. Our topology consists of 6 NFPs. Each NFP has 25 routers and 8 DCs with two racks and 20 servers per rack in each DC. A server has a total processing capacity of 10 GHz and an access link with 1 Gbps. A rack has an aggregation link with 20 Gbps capacity. For each NFP, we randomly assign BW and CPU unit costs (i.e., C_{CPU},C^{dc}_{BW} and C^{S}_{BW} in Eq. 3) out of a uniform distribution based on the parameters used in

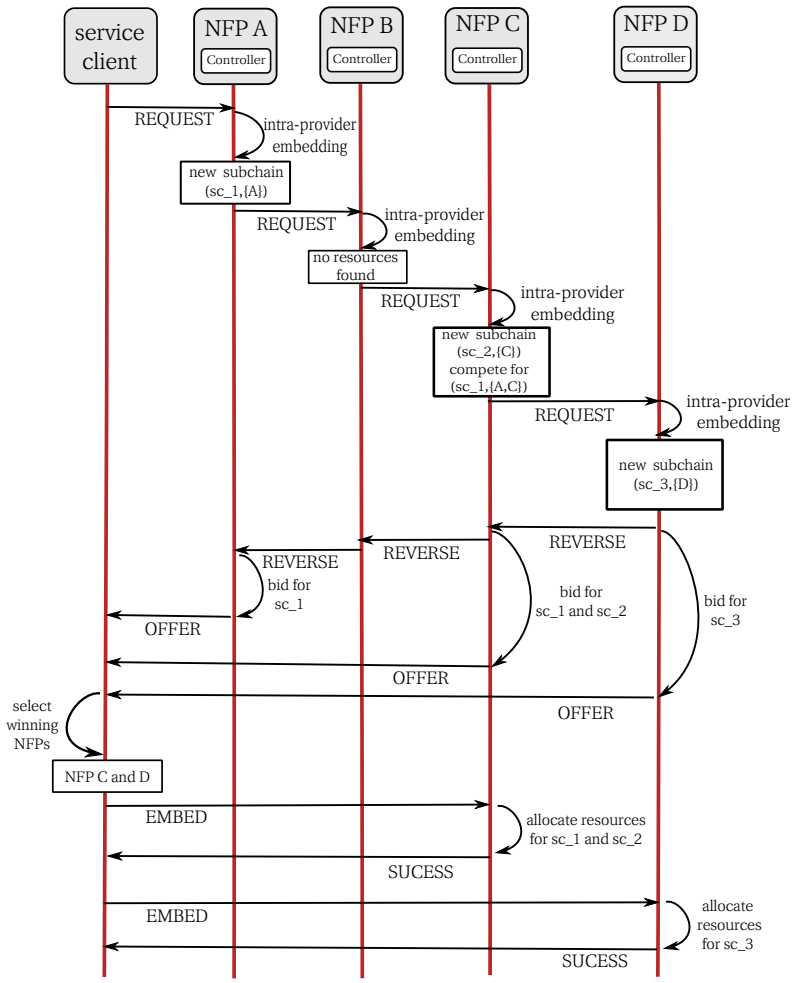


Figure 38: DistNSE protocol workflow.

[62]. Service chain requests are generated with 5 to 10 NFs with computation demand per NF and BW demand sampled out of a uniform distribution. For each service chain, we consider the shortest AS-path spanning all NFPs, and accordingly, we randomly select a source and destination router. For each test, we perform 20 runs and report the average values. We compare DistNSE against Polyvine [54], which is a widely known approach for distributed virtual network embedding. However, since Polyvine only supports virtual networks and does not provide a solution for intra-domain embedding, we adapt it to service chain embedding and augment it with our intra-provider algorithm. For both Polyvine and DistNSE, we implement service cost minimization as the local embedding policy of each NFP.

We start by measuring the total service cost for each request, which represents the total cost of all subchains (calculated with Eq. 3) of a service chain. We consider requests that are accepted by both DistNSE and Polyvine. Figure 39 shows that DistNSE incurs significantly lower

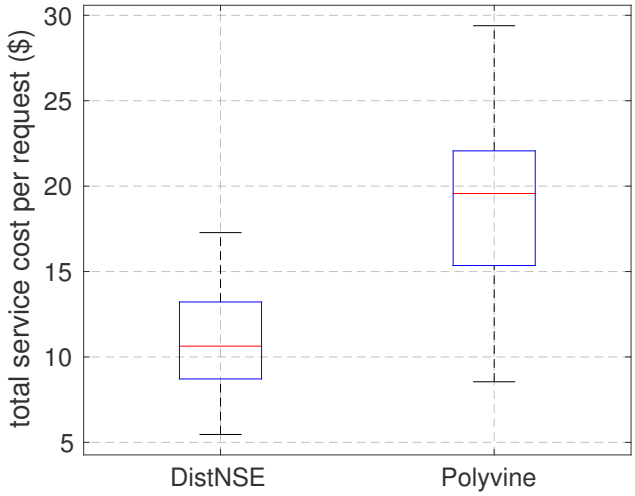


Figure 39: Total service cost per request.

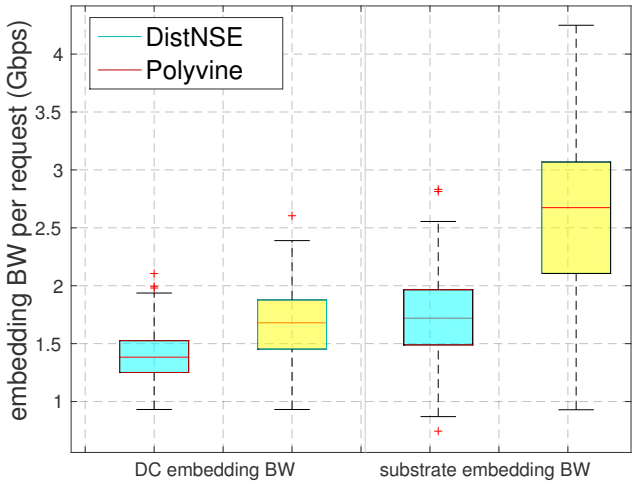


Figure 40: Embedding BW per request on substrate and DCs links.

service cost (approx. 40% less cost) than Polyine. This result corroborates the efficiency of DistNSE which enables NFPs to bid and compete for different subchains in comparison to Polyvine, which assigns each subchain to the first NFP willing to embed it.

To discover the origin of the cost difference, we further examine the embedding cost in terms of CPU and BW capacity consumed to embed a request. Our results (not shown here) show no difference in terms of CPU embedding cost because the amount of CPU required to embed an NF does not vary across NFPs. However, as illustrated in Figure 40, DistNSE incurs low BW cost on the substrate and DC network. Since a BW demand is constant (Eq. 3), this difference stems from embedding requests on shorter paths with DistNSE compared to Polyvine. This is also confirmed by the scatter plot in Figure 41 which shows a strong correlation (coefficient $r = 0.85$) between the

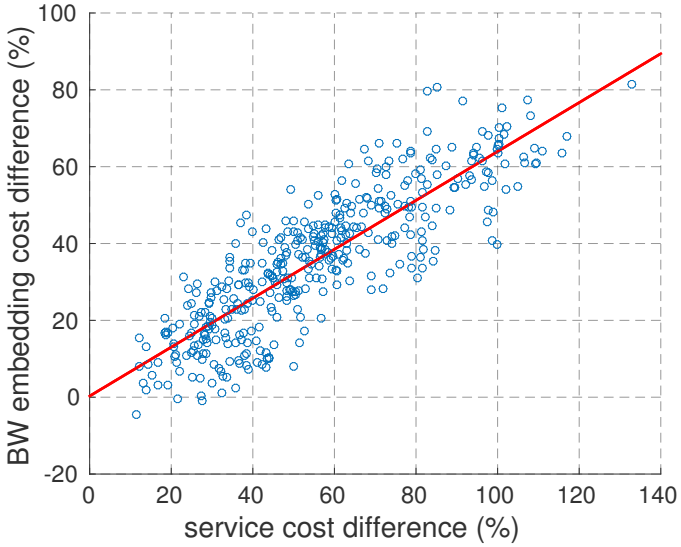


Figure 41: Service cost difference vs. BW embedding cost difference for each request.

service cost difference, when comparing DistNSE to Polyvine, and the difference in BW cost. This is further validated by the coefficients generated by the linear regression model, i.e., slope $\beta = 0.63$ and $R^2 = 0.73$. In addition, we examine the relation between the difference in service cost and the difference in the number of NFPs assigned to embed a request. Our results (not reported here) show a weak correlation ($r = 0.31$).

Since we define enabling competition as a requirement for efficient NSE, we measure the number of NFPs competing for embedding a service chain with DistNSE. We consider non-expiring requests, and to account for the number of non-competing NFPs, we run tests only on paths with 5 NFPs. Figure 42 illustrates that the number of competing NFPs decreases as more requests arrive in the network, since NFPs run out of resources to compete for a service chain (Figure 45). This is further validated through Figure 43, which shows a strong negative correlation between the number of competing NFPs and the total CPU utilization in DCs. This eventually indicates that DistNSE enables NFPs to compete equally for all service chains as long as they have sufficient resources for network service embedding.

We also measure the total number of messages exchanged to embed a service chain. According to Figure 44, the number of messages drops as more requests arrive. As depicted in Figures 42 and 45, as a smaller number of NFPs is willing compete for a service chain (because they run out of resources), fewer messages are generated. The drop in the number of messages stems from the drop in the number of OFFER, EMBED and SUCCESS messages exchanged between the

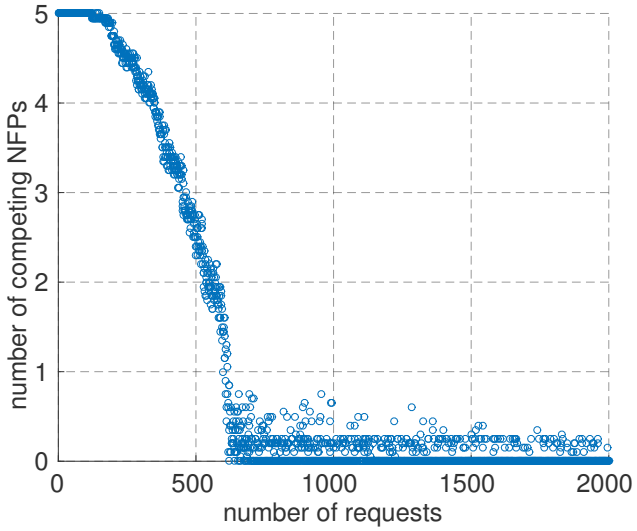


Figure 42: Number of NFPs competing for a service chain vs number of requests.

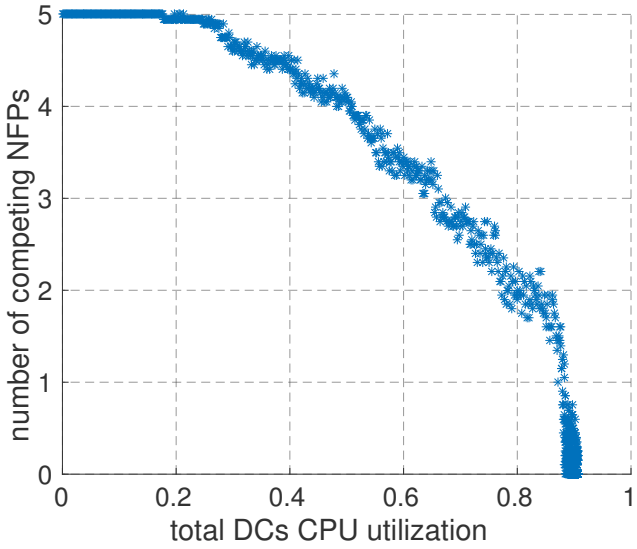


Figure 43: Number of NFPs competing for a service chain vs. the total DCs' CPU utilization.

client and the NFPs. These messages represent the participation of NFPs on bidding for a service chain (see Section 6.4).

6.6 RELATED WORK

Most existing NSE approaches have mainly focused on mapping NF service chains in DCs. CloudNaaS [43] and STRATOS [72] present heuristic algorithms to assign NFs to servers and links within DCs with the goal of minimizing inter-rack traffic. For example, Oktopus

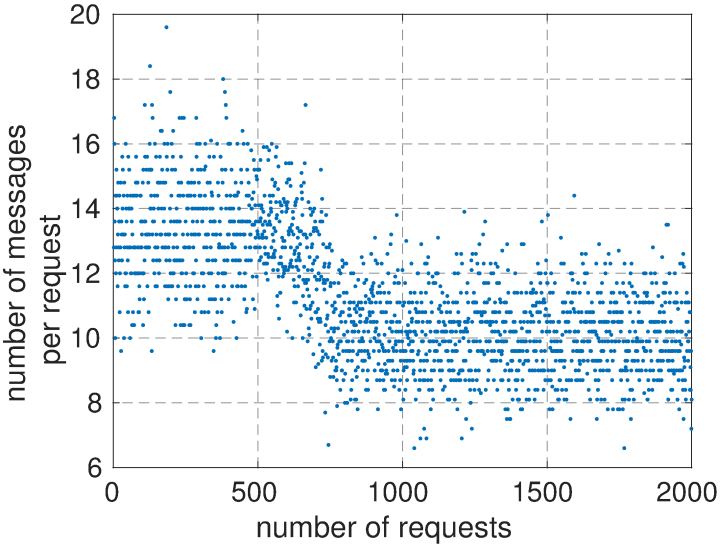


Figure 44: Number of protocol messages exchanged per service chain.

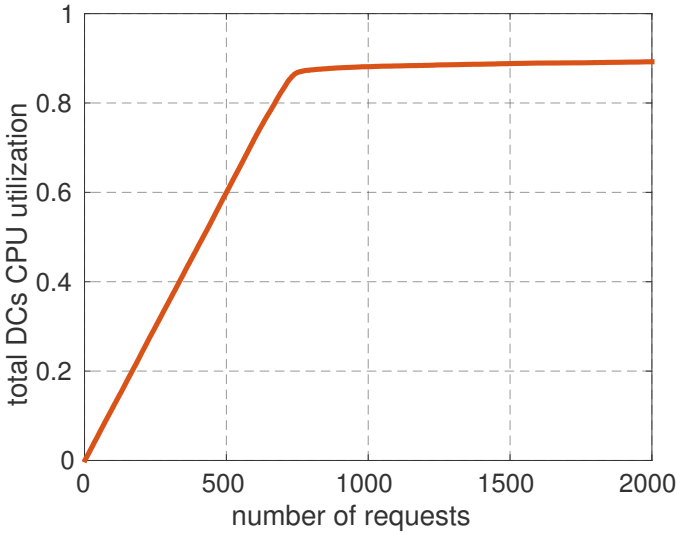


Figure 45: Total DCs' CPU utilization.

[41], SecondNet [84] and CloudMirror [102] aim at embedding virtual clusters in DCs' networks. The work in [92] provides NFs embedding along the data path assuming the deployment of processing platforms in the network and full information disclosure among the NFPs (in terms of utilization and offered processing services per node). On the other hand, Nestor [62] is a recent approach which provides NSE across DCs and NFPs by employing a DC-level abstract view of the NFP network and a centralized coordinator that partitions service chains among DCs. We propose a distributed NSE architecture which maps service chains across DCs and NFPs. Our architecture eliminates the need for a centralized coordinator, leading to better robustness (i.e., avoid a single point of failure) and to preserve both NFPs' privacy (i.e., avoid sharing NFP information with a third party) and NFPs' autonomy (i.e., NFPs can implement policies independently of each other).

There has been a large body of work on virtual network embedding [134, 135, 55, 61, 90]. However, these embedding techniques are developed to map arbitrary virtual network topologies onto substrate networks. As such, they are not optimized for service chain partitioning across NFPs and chain mapping onto DC networks. Furthermore, they usually assume a centralized controller or coordinator to perform embedding across the providers and enforce a single global embedding policy. On the other hand, Polyvine [54] is a distributed virtual network embedding framework which preserves NFPs policy and autonomy. Yet, due to the arbitrary structure of virtual networks, it does not allow different NFPs to compete for the same part of a virtual network on the same AS-path and also does not maintain the order of NFs in the chain. In contrast, we design embedding methods that optimize the mapping of service chains onto DC networks. In particular, through smart service chain partitioning, we enable different NFPs to compete for different parts of a service chain while preserving NFP's privacy and autonomy and service chain correctness.

6.7 SUMMARY

In this chapter, we have presented DistNSE, a distributed architecture that enables the collaboration among NFPs to provide NSE, while maintaining the privacy and the autonomy of each provider and ensuring competitive service pricing for the client. In particular, we have proposed a distributed algorithm and a communication protocol for partitioning service chains across different NFPs and establishing privacy-preserving competition. For intra-provider embedding, we have developed a two-stage algorithm for DC and path selection and NFs to DC mapping.

Using our simulator, we have shown that DistNSE incurs significantly low service cost per request for the client. We have further uncovered that the efficiency of DistNSE stems from establishing competition among NFPs and from consuming fewer BW units by selecting shorter embedding paths in DCs and substrate networks. Next, we have shown that DistNSE enables all NFPs on an AS-path to compete for all service chains as long as they have sufficient resources. Finally, we have quantified the number of messages generated by our protocol. Our results illustrate that the number of generated messages is proportional to the number of NFPs willing to bid for a service chain.

7 CONCLUSIONS

NFV is a recent trend aiming at replacing special-purpose and hardware-based middleboxes with software NFs that can be consolidated on processing platforms in enterprise networks and datacenters. NFV opens the door for new service models such as NFaaS, allowing the deployment of NFs in a pay-per-use fashion, while enabling on-demand resource provisioning and leading to operation and investment cost saving. Yet, the realization of NFV raises serious requirements in terms of network service deployment. In particular, NFV requires the development of techniques and approaches for resource discovery and allocation to place NFs on the NFPs' networks and datacenters. This deployment can take place on the traffic path to avoid redirection and latency inflation or off the path to serve NFs with high computational requirements. To this end, we have addressed in this thesis the problem of network service deployment and presented techniques for on and off path flow processing. We have further augmented our approaches with a workload profiling technique to gauge the computational requirements of NFs. Considering that middleboxes (e.g., firewalls, NATs, proxies) are known for their undesirable implications on traffic and for hindering connection establishment, we have also developed an SDN architecture for invariant preserving middlebox traversal. In the following, we highlight the conclusions of this thesis and provide possible directions for future work.

We have developed MIDAS, an architecture for the coordination of on-path processing setup to circumvent the difficulty in middlebox discovery and selection across multiple NFPs, without any prior knowledge of the network path. We have proposed signaling protocols for middlebox discovery and NFP interoperability and presented an algorithm for order-preserving middlebox selection within each NFP. For NFP assignment, we have leveraged on MPC to preserve the confidentiality of middleboxes utilizations across NFPs. Using our prototype implementation, we have shown that MIDAS incurs processing setup delays in the order of tenths of milliseconds. In particular, middlebox discovery and controller chaining incur low delays with a large number of middleboxes and high request arrival rates. Our experimental results show that MPC-based NFP assignment accounts for most of the processing setup delay. Nevertheless, MPC does not introduce a scalability limitation because the MPC delay is not affected by the request arrival rate and does not significantly increase with a number of NFPs that does not exceed the average

AS-length of Internet paths. Coupling MPC with our middlebox selection algorithm results in network-wide load balancing and high request acceptance rates, as shown in our simulation results.

We have presented DistNSE, a distributed architecture that enables the collaboration among NFPs to provide off-path NSE, while maintaining the privacy and the autonomy of each provider and ensuring competitive service pricing for the client. In particular, we have proposed a distributed algorithm and a communication protocol for partitioning service chains across different NFPs and establishing privacy-preserving competition. For intra-provider embedding, we have developed an algorithm for DC and path selection and NFs to DC mapping. Using our simulator, we have shown that DistNSE incurs significantly low service cost per request for the client. We have further uncovered that the efficiency of DistNSE stems from establishing competition among NFPs and from consuming fewer of BW units by selecting shorter embedding paths in DCs and substrate networks. Next, we have shown that DistNSE enables all NFPs on an AS-path to compete for all service chains as long as they have sufficient resources. Finally, we have measured the number of messages generated by our protocol. Our results illustrate that the number of generated messages is proportional to the number of NFPs willing to bid for a service chain.

Given that both DistNSE and MIDAS require the knowledge of NFs computational requirements to assign NFs' to processing platforms (i.e., servers), we have exemplified methods to circumvent the difficulty of profiling packet processing workloads on commodity servers. We have discussed the implications of polling and batch processing on workload profiling, and we further have shown that workload profiling can produce inaccurate results when the confluence of these two factors is not taken into account. Our experimental results demonstrate that our workload profiling technique can measure the computational requirements of various workloads for a wide range of packet forwarding rates. We believe that our workload profiling method can comprise a prominent component of a modern packet processing system, improving its ability to perform admission control and utilize the computing resources more efficiently.

Driven by ISPs' needs and the emergence of NFV, more middleboxes will be deployed in the network. Yet, middleboxes are known for their undesirable effect on traffic and for hampering connection establishment. In this respect, we have presented an SDN architecture for establishing invariant-preserving connections traversing middleboxes and fostering the collaboration between end-hosts and ISPs. In particular, an end-host can express a desirable behavior from the network, specified as an invariant (e.g., no IP header or payload modification), and the ISP, in turn, can establish a connection through middleboxes that preserve this invariant. To this end, we have developed an

algorithm to select redirection paths through a sequence of invariant-preserving middleboxes while considering network and middlebox utilization. Our algorithm can be adapted to fulfill different objectives: load balancing or delay minimization. Using simulations, we showed that our algorithm substantially increases the number (more than 40%) of established connections with invariant preservation and achieves a network-wide load balance as well as high network and middleboxes utilization.

7.1 FUTURE WORK

The results presented in this thesis outline several possible directions for future work. These directions can be seen within the context of NFV orchestration and management. In particular, while our network service deployment approaches can constitute a primary component to orchestrate the instantiation and deployment of NFs, other aspects still need to be addressed for NFV orchestration and management:

- While this thesis addresses NFs assignment with prior knowledge of NFs' BW and computational requirements, clients might not have this information in hand when requesting a network service. Furthermore, clients might aim to dynamically scale up or down the resources of particular NFs to cope with varying demands or load (e.g., the load of a client's Intrusion Prevention System (IPS) might increase abruptly due to a Denial-of-Service attack). In this respect, a potential direction is to extend our network service deployment techniques to support multi-provider dynamic resource allocation and provisioning for NFs deployment. These techniques should aim at minimizing provisioning time to avoid service disruption and ensure efficient functionality (e.g., the IPS resources should be scaled up in the shortest possible time to minimize Denial-of-Service attack damages such as BW consumption and down time), while preserving the correctness of the service chain and the privacy and the autonomy of the providers. This, in turn, raises the need to design online profiling and measurement methods to dynamically monitor NFs' load and deliver accurate and up-to-date readings on the NFs' state to the NFV orchestrator.

- Decoupling state from the processing is another future direction for NFV orchestration. A recent work has investigated this approach for NAT to support better scalability and resilience [95]. This can be explored further in the context of network service deployment. With the decoupling of its state, an NF will consist of two or more components (e.g., processing, cache, state) which can be deployed on different locations of the network. For instance, the state component can be deployed off-

path in datacenters, whereas the processing component can be hosted on-path on programmable routers. Furthermore, an NF can be scaled up or down by instantiating multiple processing instances, while preserving consistency through a shared state component. This provides a higher degree of flexibility and scalability to service deployment. However, this also raises a set of challenges. For instance, the deployment of the decoupled NFs should maintain the efficiency of the original NFs in terms of response time and functionality. In particular, the communication delay between the processing and the state components should not degrade the performance of the NF (e.g., an IDS should still be able to detect attacks in time to alert the network administrator). Furthermore, we need to ensure consistent and up-to-date state among the different processing instances sharing a single state instance.

BIBLIOGRAPHY

[1] Intel Data Plane Development Kit (DPDK). URL http://dpdk.org/.

[2] IGen Network Topology Generator. URL http://informatique.umons.ac.be/networks/igen.

[3] Internet2. URL http://www.internet2.edu/.

[4] Virtual Ideal Functionality Framework. URL http://www.viff.dk/.

[5] Openonload, 2008. URL http://www.openonload.org/.

[6] Intel VTune Amplifier XE 2011, 2013. URL http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[7] MCP-7002: APPLICATION DELIVERY NETWORK- A GLOBAL STRATEGIC BUSINESS REPORT, 2015. URL http://www.strategyr.com/pressMCP-7002.asp.

[8] Arista Networks, 2015. URL https://www.arista.com/en/.

[9] Internet AS-level Topology Archive, 2015. URL http://irl.cs.ucla.edu/topology/.

[10] Amazon EC2, 2015. URL https://aws.amazon.com/ec2/instance-types/.

[11] BRO, 2015. URL https://www.bro.org/.

[12] Cisco Nexus 7000 Series Switches, 2015. URL http://www.cisco.com/c/en/us/products/switches/nexus-7000-series-switches/index.html.

[13] ETSI Network Function Virtualization, 2015. URL http://www.etsi.org/technologies-clusters/technologies/nfv.

[14] Floodlight: a Java-based OpenFlow Controll, 2015. URL http://www.projectfloodlight.org/floodlight/.

[15] ITU ICT facts and figures, 2015. URL http://www.itu.int/en/ITU-D/Statistics/Pages/facts/default.aspx.

[16] IPtables, 2015. URL http://linux.die.net/man/8/iptables.

[17] LXC, 2015. URL https://linuxcontainers.org/.

[18] Open MUL: SDN Openflow Controller, 2015. URL http://sourceforge.net/p/mul/wiki/Home/.

[19] NetFPGA, 2015. URL http://netfpga.org/.

[20] OCTEON Multi-Core Processor Family, 2015. URL http://caviumnetworks.com/OCTEON_MIPS64.html.

[21] ONOS: An Open Source Distributed SDN OS, 2015. URL http://onosproject.org/.

[22] OPNFV, 2015. URL https://www.opnfv.org/.

[23] OpenDaylight: A Linux Foundation Collaborative Project, 2015. URL https://www.opendaylight.org/.

[24] OpenVZ, 2015. URL https://openvz.org/.

[25] A Python-based OpenFlow Controller, 2015. URL http://www.noxrepo.org/pox/about-pox/.

[26] QEMU, 2015. URL www.qemu.org/.

[27] PF_RING DNA, 2015. URL http://www.ntop.org/products/packet-capture/pf_ring/.

[28] Ryu: Component-based Software Defined Networking Framework, 2015. URL http://osrg.github.io/ryu/.

[29] Software-Defined Networking: The New Norm for Networks, 2015. URL https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[30] SDN Architecture Overview, 2015. URL https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf.

[31] SNORT, 2015. URL http://www.snort.org/.

[32] Docker, 2015. URL www.docker.com.

[33] squid, 2015. URL http://www.squid-cache.org/.

[34] T-NOVA Network Functions as-a-Service over Virtualised Infrastructures, 2015. URL http://www.t-nova.eu/.

[35] Unify: UNIFYing Carrier and Cloud Networks, 2015. URL https://www.fp7-unify.eu/.

[36] The Virtualization Journey, 2015. URL http://www.datacenterjournal.com/virtualization-journey/.

[37] VirtualBox, 2015. URL https://www.virtualbox.org.

[38] Understanding Full Virtualization, Paravirtualization and Hardware assist, 2015. URL http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf.

[39] Vmware Workstation, 2015. URL https://www.vmware.com/products/workstation.

[40] A. Abujoda and P. Papadimitriou. MIDAS: Middlebox Discovery and Selection for On-Path Flow Processing. In *Communication Systems and Networks (COMSNETS), 2015 7th International Conference on*, pages 1–8, Jan 2015. doi: 10.1109/COMSNETS.2015.7098686.

[41] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0797-0. doi: 10.1145/2018436.2018465. URL http://doi.acm.org/10.1145/2018436.2018465.

[42] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945462. URL http://doi.acm.org/10.1145/1165389.945462.

[43] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 8:1–8:13, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038924. URL http://doi.acm.org/10.1145/2038916.2038924.

[44] A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig. NATBLASTER: Establishing TCP Connections Between Hosts Behind NATs. In *in proceedings of ACM SIGCOMM Asia Workshop*, 2005.

[45] P. Bogetoft, D. Christensen, I. Damgard, M. Geisler, T. Jakobsen, M. Kroigaard, J. Nielsen, J. Nielsen, K. Nielsen, J. Pagter, Michael S., and T. Toft. Secure Multiparty Computation Goes Live. In *Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03548-7. doi: 10.1007/978-3-642-03549-4_20. URL http://dx.doi.org/10.1007/978-3-642-03549-4_20.

[46] F. Botelho, A. Bessani, F.M.V. Ramos, and P. Ferreira. On the Design of Practical Fault-Tolerant SDN Controllers. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 73–78, Sept 2014. doi: 10.1109/EWSDN.2014.25.

[47] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP). RFC 2205. URL https://tools.ietf.org/html/rfc2205.

[48] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134 vol.1, Mar 1999. doi: 10.1109/INFCOM.1999.749260.

[49] Z. Cai, A. L. Cox, and T. S. E. Ng. Maestro: A System for Scalable OpenFlow Control. Technical report, 2011.

[50] B. Carpenter. Architectural Principles of the Internet. RFC 1958, . URL http://www.rfc-editor.org/rfc/rfc1958.txt.

[51] B. Carpenter. Middleboxes: Taxonomy and Issues. RFC 3234, . URL http://tools.ietf.org/html/rfc3234.

[52] R. Chakravorty, S. Katti, J. Crowcroft, and I. Pratt. Flow Aggregation for Enhanced TCP over Wide-Area Wireless. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1754–1764 vol.3, March 2003. doi: 10.1109/INFCOM.2003.1209198.

[53] Sumi Choi, J. Turner, and T. Wolf. Configuring Sessions in Programmable Networks. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 60–66 vol.1, 2001.

[54] M Chowdhury, F Samuel, and R Boutaba. PolyViNE: Policy-based Virtual Network Embedding Across Multiple Domains. In *Proceedings of the Second ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, VISA '10, pages 49–56, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0199-2. doi: 10.1145/1851399.1851408. URL http://doi.acm.org/10.1145/1851399.1851408.

[55] N.M.M.K. Chowdhury, M.R. Rahman, and R. Boutaba. Virtual Network Embedding with Coordinated Node and Link Mapping. In *INFOCOM 2009, IEEE*, pages 783–791, April 2009.

[56] I. Cooper and J. Dilley. Known HTTP Proxy/Caching Problems. RFC 3143. URL https://tools.ietf.org/html/rfc3143#ref-2.

[57] I. Damgard, M. Geisler, M. Kroigaard, and J. Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *Public Key Cryptography - PKC 2009*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-00467-4. doi: 10.1007/978-3-642-00468-1_10. URL http://dx.doi.org/10.1007/978-3-642-00468-1_10.

[58] L. Deri. nCap: Wire-speed Packet Capture and Transmission. In *End-to-End Monitoring Techniques and Services*, pages 47–55, May 2005. doi: 10.1109/E2EMON.2005.1564468.

[59] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing Middlebox Interference with Tracebox. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 1–8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1953-9. doi: 10.1145/2504730.2504757. URL `http://doi.acm.org/10.1145/2504730.2504757`.

[60] A. Dhamdhere and C. Dovrolis. Twelve Years in the Evolution of the Internet Ecosystem. *IEEE/ACM Trans. Netw.*, 19(5):1420–1433, October 2011. ISSN 1063-6692. doi: 10.1109/TNET.2011.2119327. URL `http://dx.doi.org/10.1109/TNET.2011.2119327`.

[61] D. Dietrich, A. Rizk, and P. Papadimitriou. Multi-Domain Virtual Network Embedding with Limited Information Disclosure. In *IFIP Networking Conference, 2013*, pages 1–9, May 2013.

[62] D. Dietrich, A. Abujoda, and P. Papadimitriou. Network Service Embedding Across Multiple Providers with Nestor. In *IFIP Networking Conference (IFIP Networking)*, pages 1–9, May 2015. doi: 10.1109/IFIPNetworking.2015.7145312.

[63] M. Djatmiko, D. Schatzmann, X. Dimitropoulos, A. Friedman, and R. Boreli. Federated Flow-based Approach for Privacy Preserving Connectivity Tracking. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 429–440, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2101-3. doi: 10.1145/2535372.2535388. URL `http://doi.acm.org/10.1145/2535372.2535388`.

[64] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629578. URL `http://doi.acm.org/10.1145/1629575.1629578`.

[65] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 141–154, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu`.

[66] K. Egevang and K. Egevang. The IP Network Address Translator (NAT). RFC 1631. URL http://www.mpi-sws.org/~francis/rfc1631.txt.

[67] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Forwarding Path Architectures for Multicore Software Routers. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 3:1–3:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0467-2. doi: 10.1145/1921151.1921155. URL http://doi.acm.org/10.1145/1921151.1921155.

[68] R. Ennals, R. Sharp, and A. Mycroft. Task Partitioning for Multicore Network Processors. In Rastislav Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25411-9. doi: 10.1007/978-3-540-31985-6_6. URL http://dx.doi.org/10.1007/978-3-540-31985-6_6.

[69] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network Configuration Protocol (NETCONF). RFC 6241, 2011. URL https://tools.ietf.org/html/rfc6241.

[70] J. L. Eppinger. TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem. Technical report, 2005.

[71] D. Erickson. The Beacon OpenFlow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491189. URL http://doi.acm.org/10.1145/2491185.2491189.

[72] A. Gember et al. Stratos: Virtual Middleboxes as First-Class Entities. Technical report, 2013.

[73] D. Dean et al. The Internet Economy in the G-20, 2012. URL https://www.bcg.com/documents/file100409.pdf.

[74] L. Deri et al. Improving Passive Packet Capture: Beyond Device Polling. In *In Proceedings of SANE*, 2004.

[75] N. Egi et al. Scaling Middleboxes through Network-Wide Flow Partitioning. In *Proc. USENIX OSDI*, October 2010.

[76] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. FlowTags: Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 19–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491203. URL http://doi.acm.org/10.1145/2491185.2491203.

[77] S. K. Fayazbakhsh, L. C., V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 543–546, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/fayazbakhsh.

[78] B. Frank, I. Poese, Y. Lin, G. Smaragdakis, A. Feldmann, B. Maggs, J. Rake, S. Uhlig, and R. Weber. Pushing CDN-ISP Collaboration to the Limit. *SIGCOMM Comput. Commun. Rev.*, 43(3): 34–44, July 2013. ISSN 0146-4833. doi: 10.1145/2500098.2500103. URL http://doi.acm.org/10.1145/2500098.2500103.

[79] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-defined Middlebox Networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 7–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1776-4. doi: 10.1145/2390231.2390233. URL http://doi.acm.org/10.1145/2390231.2390233.

[80] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626313. URL http://doi.acm.org/10.1145/2619239.2626313.

[81] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy. Flow Processing and the Rise of Commodity Network Hardware. *SIGCOMM Comput. Commun. Rev.*, 39(2):20–26, March 2009. ISSN 0146-4833. doi: 10.1145/1517480.1517484. URL http://doi.acm.org/10.1145/1517480.1517484.

[82] A. Gudipati, D. Perry, L. E. Li, and S. Katti. SoftRAN: Software Defined Radio Access Network. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 25–30, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491207. URL http://doi.acm.org/10.1145/2491185.2491207.

[83] S. Guha, Y. Takeda, and P. Francis. NUTSS: A SIP-based Approach to UDP and TCP Network Connectivity. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, FDNA '04, pages 43–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-942-X. doi: 10.1145/1016707.1016715. URL http://doi.acm.org/10.1145/1016707.1016715.

[84] C. Guo, G. Lu, J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A Data Center Network Virtual-

ization Architecture with Bandwidth Guarantees. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 15:1–15:12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0448-1. doi: 10.1145/1921168.1921188. URL http://doi.acm.org/10.1145/1921168.1921188.

[85] Debayan Gupta, Aaron Segal, Aurojit Panda, Gil Segev, Michael Schapira, Joan Feigenbaum, Jenifer Rexford, and Scott Shenker. A New Approach to Interdomain Routing Based on Secure Multi-party Computation. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 37–42, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1776-4. doi: 10.1145/2390231.2390238. URL http://doi.acm.org/10.1145/2390231.2390238.

[86] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206, August 2010. ISSN 0146-4833. doi: 10.1145/1851275.1851207. URL http://doi.acm.org/10.1145/1851275.1851207.

[87] R. Hancock, G. Karagiannis, J. Loughney, and S. Van den Bosch. Next Steps in Signaling (NSIS) Framework. RFC 4080, 2006. URL https://www.ietf.org/rfc/rfc4080.txt.

[88] W. Henecka and M. Roughan. STRIP: Privacy-Preserving Vector-Based Routing. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–10, Oct 2013. doi: 10.1109/ICNP.2013.6733586.

[89] V. Heorhiadi, M. K. Reiter, and V. Sekar. New Opportunities for Load Balancing in Network-wide Intrusion Detection Systems. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 361–372, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1775-7. doi: 10.1145/2413176.2413218. URL http://doi.acm.org/10.1145/2413176.2413218.

[90] Ines Houidi, Wajdi Louati, Walid Ben Ameur, and Djamal Zeghlache. Virtual Network Provisioning Across Multiple Substrate Networks . *Computer Networks*, 55(4):1011 – 1023, 2011. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/j.comnet.2010.12.011. URL http://www.sciencedirect.com/science/article/pii/S1389128610003786. Special Issue on Architectures and Protocols for the Future Internet.

[91] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity Switch Models for Software-defined Network Emulation. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 43–48, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491188. URL http://doi.acm.org/10.1145/2491185.2491188.

[92] Xin Huang, S. Ganapathy, and T. Wolf. A Scalable Distributed Routing Protocol for Networks with Data-path Services. In *Network Protocols, 2008. ICNP 2008. IEEE International Conference*, pages 318–327, Oct 2008.

[93] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-636-6. doi: 10.1145/1658939.1658941. URL http://doi.acm.org/10.1145/1658939.1658941.

[94] K. Jang, S. Han, S. Han, S-. Moon, and K. Park. Accelerating SSL with GPUs. *SIGCOMM Comput. Commun. Rev.*, 41(1):135–139, January 2011. ISSN 0146-4833. doi: 10.1145/1925861.1925885. URL http://doi.acm.org/10.1145/1925861.1925885.

[95] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless Network Functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 49–54, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3540-9. doi: 10.1145/2785989.2785993. URL http://doi.acm.org/10.1145/2785989.2785993.

[96] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[97] J. Kelly, W. Araujo, and K. Banerjee. Rapid Service Creation Using the JUNOS SDK. *SIGCOMM Comput. Commun. Rev.*, 40 (1):56–60, January 2010. ISSN 0146-4833. doi: 10.1145/1672308.1672320. URL http://doi.acm.org/10.1145/1672308.1672320.

[98] Wookyun Kho, S.A. Baset, and Henning Schulzrinne. Skype Relay Calls: Measurements and Experiments. In *INFOCOM Workshops 2008, IEEE*, pages 1–6, April 2008.

[99] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000. ISSN 0734-2071. doi: 10.1145/354871.354874. URL http://doi.acm.org/10.1145/354871.354874.

[100] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and

S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924968.

[101] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *CoRR*, abs/1406.0440, 2014. URL http://arxiv.org/abs/1406.0440.

[102] J. Lee, M. Lee, L. Popa, Y. Turner, S. Banerjee, P. Sharma, and B. Stephenson. CloudMirror: Application-Aware Bandwidth Reservations in the Cloud. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, San Jose, CA, 2013. USENIX. URL https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/Lee.

[103] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically Centralized?: State Distribution Trade-offs in Software Defined Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1477-0. doi: 10.1145/2342441.2342443. URL http://doi.acm.org/10.1145/2342441.2342443.

[104] H. Liao, C. R. Lin, Y. Lin, and K. Tung. Intrusion Detection System: A Comprehensive Review. *Journal of Network and Computer Applications*, 36(1):16 – 24, 2013. ISSN 1084-8045. doi: http://dx.doi.org/10.1016/j.jnca.2012.09.004. URL http://www.sciencedirect.com/science/article/pii/S1084804512001944.

[105] J. Manner, G. Karagiannis, and A. McDonald. NSIS Signaling Layer Protocol (NSLP) for Quality-of-Service Signaling. RFC 5974. URL https://tools.ietf.org/html/rfc5974.

[106] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins.

[107] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. ISSN 0146-4833. doi: 10.1145/1355734.1355746. URL http://doi.acm.org/10.1145/1355734.1355746.

[108] B.A.A. Nunes, M. Mendonca, Xuan-Nam Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *Communications Surveys Tutorials, IEEE*, 16(3):1617–1634, Third 2014. ISSN 1553-877X. doi: 10.1109/SURV.2014.012214.00180.

[109] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047, 2013. URL `https://tools.ietf.org/html/rfc7047`.

[110] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[111] Z. Ayyub Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486022. URL `http://doi.acm.org/10.1145/2486001.2486022`.

[112] P. Quinn and T. Nadeau. Problem Statement for Service Function Chaining. RFC 7498, 2015. URL `https://tools.ietf.org/html/rfc7498`.

[113] C. Raiciu, V. Olteanu, and R. Stoenescu. Good Cop, Bad Cop: Forcing Middleboxes to Cooperate. In *IAB Workshop on Stack Evolution in a Middlebox Internet (SEMI)*, IAB. IAB, 2015.

[114] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168. URL `https://tools.ietf.org/html/rfc3168`.

[115] L. Rizzo, M. Carbone, and G. Catalli. Transparent Acceleration of Software Packet Forwarding Using Netmap. In *INFOCOM, 2012 Proceedings IEEE*, pages 2471–2479, March 2012. doi: 10.1109/INFCOM.2012.6195638.

[116] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN-Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489. URL `https://tools.ietf.org/html/rfc3489`.

[117] Frank La Rue. Report of the Special Rapporteur on the Promotion and Protection of the Right to Freedom of Opinion and Expression, 2015. URL `http://www2.ohchr.org/english/bodies/hrcouncil/docs/17session/A.HRC.17.27_en.pdf`.

[118] R. Sedgewick. *Algorithms in C, Part 5: Graph Algorithms*. Addison-Wesley Professional, third edition, 2001. ISBN 9780768685329.

[119] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar.

[120] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced Study of SDN/OpenFlow Controllers. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '13, pages 1:1–1:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2641-4. doi: 10.1145/2556610.2556621. URL http://doi.acm.org/10.1145/2556610.2556621.

[121] A. Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, November 1979. ISSN 0001-0782. doi: 10.1145/359168.359176. URL http://doi.acm.org/10.1145/359168.359176.

[122] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1419-0. doi: 10.1145/2342356.2342359. URL http://doi.acm.org/10.1145/2342356.2342359.

[123] J.E. Smith and R. Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, May 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.173.

[124] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP Topologies with Rocketfuel. *IEEE/ACM Trans. Netw.*, 12 (1):2–16, February 2004. ISSN 1063-6692. doi: 10.1109/TNET.2003.822655. URL http://dx.doi.org/10.1109/TNET.2003.822655.

[125] R. Stewart. Stream Control Transmission Protocol. RFC 4960. URL https://tools.ietf.org/html/rfc4960.

[126] M. Stiemerling, J. Quittek, , and C. Cadar. NEC's simple middlebox configuration (SIMCO). RFC 4540, . URL http://tools.ietf.org/html/rfc4540.

[127] M. Stiemerling, H. Tschofenig, C. Aoun, and E. Davies. NAT/-Firewall NSIS Signaling Layer Protocol (NSLP). RFC 5973, . URL https://tools.ietf.org/html/rfc5973.

[128] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. SymNet: Static Checking for Stateful Networks. In *Proceedings of the 2013*

Workshop on Hot Topics in Middleboxes and Network Function Virtualization, HotMiddlebox '13, pages 31–36, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2574-5. doi: 10.1145/2535828.2535835. URL http://doi.acm.org/10.1145/2535828.2535835.

[129] Daniel M. Sunday. A Very Fast Substring Search Algorithm. *Commun. ACM*, 33(8):132–142, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79184. URL http://doi.acm.org/10.1145/79173.79184.

[130] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):374–385, August 2011. ISSN 0146-4833. doi: 10.1145/2043164.2018479. URL http://doi.acm.org/10.1145/2043164.2018479.

[131] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 99–112, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1972457.1972468.

[132] T. Wolf. In-Network Services for Customization in Next-Generation Networks. *Network, IEEE*, 24(4):6–12, July 2010. ISSN 0890-8044. doi: 10.1109/MNET.2010.5510912.

[133] Q. Wu and T. Wolf. Runtime Task Allocation in Multicore Packet Processing Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 23(10):1934–1943, Oct 2012. ISSN 1045-9219. doi: 10.1109/TPDS.2012.56.

[134] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration. *SIGCOMM Comput. Commun. Rev.*, 38(2):17–29, March 2008. ISSN 0146-4833. doi: 10.1145/1355734.1355737. URL http://doi.acm.org/10.1145/1355734.1355737.

[135] Y. Zhu and M. Ammar. Algorithms for Assigning Substrate Network Resources to Virtual Network Components. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, April 2006. doi: 10.1109/INFOCOM.2006.322.

CURRICULUM VITAE

PERSONAL INFORMATION

Name	**Ahmed Mohamed Ahmed Abujoda**
Date of Birth	February 15^{th} 1983
Nationality	German

WORK EXPERIENCE

06/2010 – present **Institute of Communications Technology, Leibniz Universität Hannover**
Research assistant / PhD candidate
Research areas: Network Function Virtualization (NFV), Software Defined Networking (SDN), software profiling, packet processing on commodity servers, network service embedding.

- Deployed and managed a large scale testbed for network experimentation (80+ servers, over 400 network interfaces, 6 switches, 20 NetFPGA cards). Installed FreeBSD-based testbed software comprised of various network services (NFS, DNS, Firewall, Nagios).
- Designed and implemented a SDN architecture for managing crowd-shared WMN within EU CONFINE project.
- Designed and implemented an on-path flow processing architecture within EU T–NOVA project.
- Designed and implemented an approach for profiling Click-based packet processing workloads.
- Published and presented research results in international conferences.

01/2010 – 05/2010 **Zentrum für Sensorsysteme, Universität Siegen**
Embedded development engineer

- Optimized embedded UDP/IP- Gigabit Ethernet stack implementation.
- Developed embedded networking software using C language, FPGA and PowerPC microprocessor for data acquisition.
- Implemented test software using C and visual C#.

EDUCATION

10/2007 – 12/2009 **Master of Science in Mechatronics Engineering**, Universität Siegen
Electrical engineering, Computer Science and Mechanical Engineering

10/1999 – 08/2004 **Bachelor of Science in Electrical Engineering**, Sana'a University
Electrical engineering
Major field: *Electronics and Communications*.

TECHNICAL SKILLS

Software performance profiling using Oprofile and Intel vtune.

Passive and active network measurements using tcpdump/Wireshark, traceroute, nagios, iperf, topp, and pathload.

Networking protocols (TCP/IP), software defined networks, network virtualization (OpenFlow, POX, VLAN, XEN, OpenvSwitch) and packet processing on commodity servers (Click Modular Router).

Software development and scripting using Python, C/C++, visual C#, Bash, MySQL, SQL, Visual Basic and Matlab.

Deployment and administration of testbeds, experimental networks and LAN services (Emulab, PlanetLab, Nagios).

Unix/Linux and Windows common software and systems.

LANGUAGES

Arabic	*native language*
English	*fluent*
German	*good*

Hanover, April 10, 2016

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`http://code.google.com/p/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

Final Version as of April 10, 2016 (`classicthesis` Version 0.1).