Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Design and Implementation of Patterns for Distributing Virtual Network Functions

Jan Strauß

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	Dr. rer. nat. Frank Dürr, Thomas Kohler, M.Sc.
Commenced:	March 6, 2018
Completed:	September 6, 2018

Abstract

Network functions are conventionally deployed in the form of specialized hardware appliances. With the growing demand for network agility driven by the rising adaption of virtualization and IaaS workflows, the manual deployment and configuration of physical network function appliances becomes a limiting factor. Network function virtualization (NFV) is a concept that tries to improve network flexibility by converting network functions from dedicated and proprietary hardware appliances to software that runs on general purpose servers. While losing the performance benefit of specialized appliances, NFV allows to dynamically scale and reconfigure network functions. Realizing a NFV deployment however is not trivial as there are many requirements like load balancing and high availability as well as peculiarities, related to the stateful nature of most network functions, to consider. In addition to commodity servers, there is also the possibility to run network functions directly in the network by using capabilities of modern SDN based switches. While this on-path processing could reduce latency, it also further complicates the NFV orchestration and implementation.

In this thesis, a survey of the vast body of literature that is concerned with different aspects of NFV, is presented. Based on the related work a generalized system model and design, that also supports on switch network functions, is derived and a catalog of patterns, that are applicable to problems often found in NFV systems, is compiled. Further, a prototype, that implements a subset of the presented patterns in the context of a commonly used network function, is described and the performance and fault tolerance failover behavior of the prototype were evaluated on a realistic testbed.

Kurzfassung

Netzwerkfunktionen werden herkömmlicherweise in der Form von spezialisierter Hardware eingesetzt. Mit der steigenden Nachfrage nach Netwerkflexibilität, verursacht durch den zunehmenden Einsatz von Virtualisierungs und IaaS Arbeitsabläufen, wird die manuelle Installation und Konfiguration von physischen Netzwerkfunktionen ein limitierender Faktor. Network Function Virtualization (NFV) ist ein Konzept das versucht, die Netwerkflexibilität zu verbessern, indem Netzwerkfunktionen von dedizierter und proprietärer Hardware zu Software die auf Allzweck-Servern läuft, konvertiert wird. Während der Leistungsvorteil, den spezialisierte Hardware bietet, verloren geht, erlaubt NFV es, Netzwerkfunktionen dynamisch zu skalieren und umzukonfigurieren. NFV umzusetzen ist jedoch nicht trivial, da viele Anforderungen wie Lastverteilung und Hochverfügbarkeit, als auch Besonderheiten die mit der zustandshaften Natur vieler Netwerkfunktionen zusammenhängen, zu beachten sind. Zusätzlich zu Allzweck-Servern, besteht dank der Fähigkeiten von modernen SDN Switches auch die Möglichkeit, Netzwerkfunktionen direkt im Netzwerk auszuführen. Während diese On-Path Verarbeitung Latenzen verringern könnte, würde es die Komplexität von NFV Orchestration und Implementation weiter erhöhen.

In dieser Arbeit wird eine Übersicht über die umfangreiche Literatur, die sich mit verschieden Aspekten von NFV beschäftigt, vorgestellt. Basierend auf der Literatur wird ein generalisiertes Systemmodell und Design, das auch Netzwerkfunktionen auf Switches unterstützt, abgeleitet und ein Katalog von Lösungsmustern, die bei häufig gefundenen Problemstellungen in NFV Systemen anwendbar sind, zusammengestellt. Desweiteren wird ein Prototyp, der eine Teilmenge der vorgestellten Muster im Kontext einer häufig eingesetzten Netzwerkfunktion implementiert, beschrieben und hingehend seiner Leistung und seines Verhalten in einem Failover-Fall in einer realistischen Testumgebung evaluiert.

Contents

1 Introduction and Motivation **13**
- 1.1 Contributions . . . 14
- 1.2 Thesis Structure . . . 14

2 Background **15**
- 2.1 Software Defined Networking (SDN) and OpenFlow . . . 15
- 2.2 User Space Networking . . . 16
- 2.3 Network Address Translation (NAT) . . . 16
- 2.4 Network Function Virtualization (NFV) . . . 18

3 Related Work **19**
- 3.1 Placement . . . 20
- 3.2 Chaining . . . 20
- 3.3 Consolidation . . . 21
- 3.4 Frameworks . . . 22
- 3.5 Orchestration . . . 24
- 3.6 State Management . . . 25
- 3.7 Hardware Acceleration . . . 28
- 3.8 Application Use Cases . . . 29

4 System and Entity Model **31**
- 4.1 Network Entities . . . 32
- 4.2 Network Function . . . 33
- 4.3 Network Function Chain . . . 33
- 4.4 Orchestration . . . 34

5 Design **37**
- 5.1 Network Function Properties . . . 37
- 5.2 State Management . . . 38
- 5.3 Chains, Scaling and Steering . . . 40
- 5.4 Switch Offloading . . . 42

6 Pattern Catalog **45**
- 6.1 Topological . . . 46
- 6.2 State . . . 50
- 6.3 Dynamic . . . 59
- 6.4 Implementation . . . 67

7 Implementation **71**
- 7.1 Scope . . . 71

7.2 Architecture 72
7.3 Data Model 77
7.4 State Synchronization 78

8 Evaluation **83**
8.1 Throughput 84
8.2 Latency 86
8.3 Failover 87
8.4 Server NAT Breakdown 91
8.5 Discussion 92

9 Conclusion and Future Work **95**
9.1 Future Work 95

Bibliography **97**

List of Figures

2.1 Example showing how NAT modifies traversing packets and the internal state . . 17

4.1 System model . . . 31
4.2 Network entity model . . . 32
4.3 Network function model . . . 33
4.4 Network function chain model . . . 34
4.5 Orchestration model . . . 35

5.1 Decision tree for a state management pattern . . . 39
5.2 Example chain . . . 40
5.3 Decision tree for a steering pattern . . . 41

7.1 High level overview of the implemented system . . . 73
7.2 Switch NAT architecture . . . 74
7.3 Server NAT architecture . . . 76
7.4 NAT server internal processing pipelines . . . 77
7.5 High level NAT entry representation . . . 78
7.6 State machine of the sync mechanism . . . 80
7.7 Difference between asynchronous and synchronous modes . . . 80

8.1 Testbed Topology . . . 83
8.2 NAT server pps with different flow counts . . . 84
8.3 Effect of bidirectional traffic on throughput . . . 85
8.4 NAT switch pps with different flow counts . . . 86
8.5 Split processing throughput at the server NAT and the receiver host, 1024 flows. . 87
8.6 Per packet latency for established flows, 60 bytes, 500 pps, outliers not shown (cf. Figure 8.8a) . . . 88
8.7 Per packet latency for new flows, 60 bytes, 500 pps, outliers not shown (cf. Figure 8.8b) . . . 89
8.8 CDF of per packet latency . . . 89
8.9 Histograms showing the observed packet drops during a failover event (green) and the duration of different operations during failover (blue) . . . 90
8.10 Timeline of failover events showing the packet rate at the receiver for different flow counts, 6 Mpps . . . 91
8.11 Duration of server NAT processing steps . . . 92

List of Tables

5.1 State classification 38
5.2 Network function properties 38

6.1 Patterns 45

7.1 Switch connection module events 73
7.2 Core to switch connection module events 75
7.3 State Synchronization Messages 79

1 Introduction and Motivation

In most networks, in addition to end hosts and conventional forwarding elements, e.g. switches and routers, multiple network functions perform critical tasks related to security, e.g. firewalls and intrusion detection systems, optimization, e.g. caches, load balancer, or provide additional network functionality, e.g. Network Address Translation (NAT) or tunneling. These network functions were, or still are, mostly performed by specialized, proprietary and often expensive, hardware appliances. In contrast to modern Software defined Networking (SDN) based networks, that can be configured centrally and offer a global view of the network, network function management is typically still done on a per appliance level. Considering the large number of network functions deployed in enterprise networks ([SR12] reports that the number of network functions is in the same range as number of routers) and that network functions are often not operating in isolation, but are chained together to enforce complex policies, as well as the increasingly dynamic nature of networks driven by highly dynamic workloads and new paradigms like Infrastructure as a Service (IaaS), this fine-grained management model, that requires frequent and complex human intervention, appears to be a limiting factor.

Network Function Virtualization (NFV) is an approach that, as the name suggests, virtualizes network functions, which means they are re-implemented in software, packaged in a VM or container format and will run on commodity servers. Treating network functions like a conventional compute task reduces costs, as the purchase of (often expensive) hardware appliances falls away and Virtual Network Functions (VNF) can be executed on existing compute clusters based on cheap commodity server hardware. This further allows to deploy, scale and reconfigure VNF instances on demand, which offers the required flexibility and agility to deal with IaaS workloads.

However, the implementation of a NFV solution must also account for complex requirements that are inherent to network functions and their management. The forwarding between VNF instances according to network function chains must be enforced, often involving branching and filtering while accounting for changes to the traffic and instance count. The stateful nature, for example a firewall will track TCP connection state, of many network functions further complicates forwarding, as packets must be processed by an instance that holds state associated with a packet, otherwise the correctness of a network function can be violated. Network function state also often has to be replicated, either because the state must be available on multiple instances or to enable fault tolerance. Combined with strict latency limits [SGB+15] for network functions, the implementation of either, a NFV orchestration system or a single VNF, is a complex endeavor.

While SDN can be leveraged to implement the complex forwarding of NFV systems, it might offer additional benefits for NFV. Some network functions that fit the match-action model of switches, could be implemented directly in the data plane using SDN. Open networking hardware, in particular white-box switches, became the prevalent platform to realize SDN deployments and these white-box switches typically run an open Linux based operating system as their control plane. This open OS could be used to run either, the control component of data plane capable network

functions on the same switch they are performed on, or network functions that don't fit the data plane model. This on-path processing could reduce latency and network load, but further complicates the already complex problem space of NFV orchestration and VNF implementation, due to the different execution environments and capabilities of server and switch network functions.

To this end, this thesis presents patterns, i.e. solution templates, for the orchestration and implementation of VNFs that can be executed on commodity servers and network elements.

1.1 Contributions

In particular this thesis contains the following contributions:

- An extensive survey of NFV literature, categorized by aspect.
- A NFV system model and design that supports switch offloading.
- A catalog of patterns for implementing different aspects of a NFV system, based on approaches in literature, as well as patterns for on switch processing.
- A prototype that implements a subset of the presented patterns in the context of a common network function and a focus on on switch processing.
- Evaluation of the prototype implementation, showing the performance and failover behavior.

1.2 Thesis Structure

The rest of the thesis is structured as follows:

Chapter 2 provides a short overview of concepts related to this thesis.

Chapter 3 presents a survey of NFV literature.

Chapter 4 describes the proposed system model.

Chapter 5 discusses general design considerations and problems that the patterns can be applied to.

Chapter 6 contains the pattern catalog.

Chapter 7 gives an overview of the implemented prototype.

Chapter 8 shows and discusses the evaluation results of the prototype implementation.

Chapter 9 finally gives a conclusion and describes possible future work.

2 Background

This chapter will give a brief overview of concepts related to this thesis.

2.1 Software Defined Networking (SDN) and OpenFlow

Software Defined Networking (SDN) is an approach to (mainly data-center) networking that separates the packet forwarding (data plane) from the decision on how to forward packets (control plane). The control plane is centralized in the so called controller[1] application, providing a global view of the whole network and single configuration endpoint. The global view and high level API improve network agility, visibility and make it easy to develop new network features. All these features are realized without sacrificing the high performance packet forwarding of the data plane. The forwarding in the data plane is, as in conventional networks, done by specialized switching ASICs.

SDN is often implemented with so called white box switches. These switches are built from inexpensive, off the shelf hardware and support different network operation systems (NOS), eliminating the vendor lock in of conventional hardware.

OpenFlow is the de-facto standard protocol between the control and data plane entities in SDN networks. OpenFlow defines am abstraction for programming forwarding elements via flow tables, statistics and how forwarding elements notify the control plane about events. Flow tables contain flow rules, which are defined by a filter and a list of actions. The filter defines (partial) header field values (e.g. source/destination MAC/IP addresses or TCP/UDP ports), that incoming packets are matched against. The flow rules are ordered by a priority and the first matching rules action list is applied to the packet. Possible actions include forwarding to another table, dropping, forwarding on a certain port, modification of header values or forwarding to the controller(s). A relativly new additon in this space is the P4 network programming language [BDG+14], claiming greater flexibility than OpenFlow. While in OpenFlow the table filters are limited by the header fields OpenFlow supports, P4 allows to define what the switch should interpret as header fields, enabling completely new protocols to be implemented.

[1] While logically centralized, controllers are often physically distributed to improve availability.

2.2 User Space Networking

Typical server network cards operate at 10 or 40 Gbit/s, yet application using the Linux kernel network stack only reach a fraction of these speeds [Riz12]. While the network stack of the Linux kernel and the conventional socket APIs are proven and flexible, performance wise there are several negative aspects. Memory is often dynamically allocated per-packet, frequent system calls and context switches have a significant overhead and packets are copied multiple times.

To process 64-byte packets at 10 Gbit/s, only 67.2ns are available per packet [lwn15]. Considering that a cache miss adds about 32 ns latency, a system call about 42 ns and each lock/unlock of a spinlock atleast 8.25 ns, this discrepancy between hardware capability and observed application performance comes at no surprise. While the kernel developers continue to improve the kernel performance (e.g. eBPF and XDP), there exist several solutions that bypass the kernel for high performance packet processing. DPDK [dpdk18], alongside Netmap [Riz12] are the most prominent user space networking frameworks. DPDK uses an UIO driver to allocate ring buffers in user space using huge pages, copying packets directly from/to the NIC queues to/from the user space buffers. Packets are processed in batches to further reduce the per-packet overhead, workloads are assigned to a fixed core that is busy polling the NIC and the ring buffers are implemented without locks. These frameworks make it possible to saturate a 10 Gbit/s link with a single core, depending on the application. The APIs of these frameworks, especially DPDK, are more complex than the normal socket API (and only offer acces to the raw ethernet frames, all protocols have to be implemented by the application), but there are frameworks built on top of DPDK, for example mTCP [JWJ+14] which offers a drop-in replacement for Berkeley socket API.

2.3 Network Address Translation (NAT)

Network address translation (NAT) is a technique used to map a set of network addresses to another set of addresses. The most prevalent use case of NAT today is to connect a network that uses a private IPv4 subnet (10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16), to the public internet via a single public IPv4 address. This method is often used by ISPs as a means of dealing with the increasing exhaustion of the IPv4 address space [TW10, p. 451-454].

In order to replace the internal private IP addresses it it not sufficient to just replace the source IP: After an internal IP was replaced, there would be no way to determine to which host a response from an external host belongs. The trick that NAT applies is to use the layer 4 UDP and TCP port numbers[2] to keep track of what host a response belongs to. NAT not only replaces the IP address, but also the source port number with a number that identifies the source and destination hosts, i.e. a single flow.

Figure 2.1 illustrates how NAT works: When an internal host sends a packet to an external IP *a)*, the NAT box will first try to find an existing mapping for the flow of the packet *b)*. If no mapping is found, a free IP, protocol, port triple is taken from the pool *c)* and mappings for the new flow based on the triple are inserted into the internal *d)* and external *e)* mapping tables. After applying the mapping to the packet, the source IP and UDP port number have been replaced with the values

[2] or similar higher level identifiers for other protocols, if supported

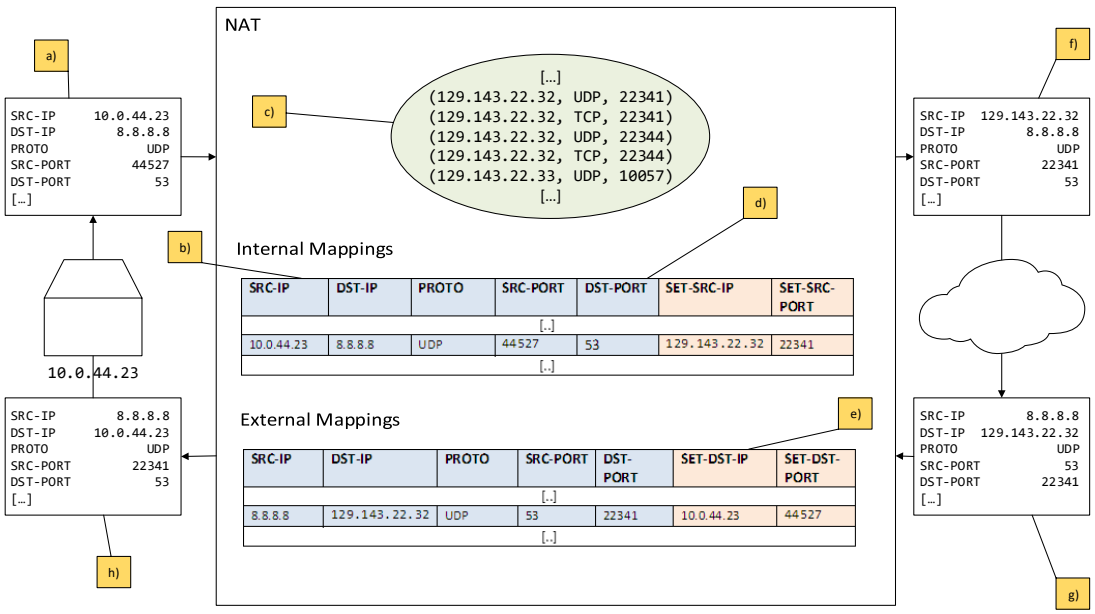


Figure 2.1: Example showing how NAT modifies traversing packets and the internal state

from the NAT pool *f)*. If the server the packet was sent to eventually replies, the packet is addressed to the replaced IP and UDP port *g)*. The NAT box will now lookup the packet's flow in the external mappings table. If no entry is found, the packet is dropped, otherwise it is rewritten according to the mapping and send to the internal network *h)* where the sending host will receive the packet.

This internal to external translation is more specifically called (dynamic) source NAT (SNAT). There is also destination NAT (DNAT) where the mappings are reversed, i.e. external packets are mapped to an internal private IP, which can be used for e.g. load balancing. There are also NAT use cases where the mappings are static instead of dynamically taken from a pool of free external addresses.

While NAT works as a solution for the IPv4 address shortage, it does violate multiple concepts of the IP layer: For example, NAT breaks the end-to-end principle, so external hosts can't connect to internal hosts if there is no mapping defined[3]. Another critical point is that a NAT box holds connection state, violating the connectionless nature of the IP layer. This state also means that if the NAT box crashes, all connections that were going through the box are killed as the mappings are lost. Moreover, NAT also breaks the separation of protocol layers, using layer 4 information to modify layer 3 headers and therefore making assumptions on what the next layer looks like, rendering any effort to introduce a new layer 4 protocol impossible, as any NAT box would have to be updated to support this new protocol. This contributes, among other factors, to a problem called internet ossification [TT05]. For a more detailed discussion of the downsides of NAT, see [TW10, p. 454-455].

[3] which, from a security standpoint, is often wanted

IPv6 solves the address shortage problem that IPv4 has, making it possible to assign an unique address to any host. However, as the slow adoption of IPv6 shows, IPv4 is likely here to stay for quite some time and therefore NAT will also be used, making it a good example of a typical, stateful network function.

2.4 Network Function Virtualization (NFV)

Network functions (NF) like firewalls, intrusion detection and prevention systems, proxies, load balancer, caches or NAT devices have traditionally been deployed as specialized and proprietary appliances. Each of those appliances had to be physically deployed to the network and manually configured, a slow and error prone process, making it hard for network administrators to keep up with the increasing demand for IaaS support in on-premise networks.

Network function virtualization (NFV) is an approach that tries to offer a more flexible, efficient and cost effective model for NF administration by replacing the conventional specialized appliances with off-the-shelf x64 servers that run the network functions, typically in VMs or containers. While the standard servers are not as performant as specialized hardware[4], the possibility to flexibly scale NFs, combined with progress in virtualization and high performance user space networking stacks, make NFV feasible.

This new model however also poses new challenges for building and running network functions. Distributed network functions require sophisticated orchestration, coordination with the network and state synchronization to guarantee correct behavior and to comply with strict uptime SLAs. The related work chapter will elaborate on the different aspects that have to be considered and solved in a NFV system.

[4] or software implementations running on dedicated appliances

3 Related Work

There is a large body of scientific work related to different aspects o fNFV, as several surveys on the topic, e.g. [LQ16; MSG+16] and [LC15], show. This chapter will give an overview of this work and provide a basic categorization based on the different aspects that NFV research is concerned with. We identified the following aspects:

Placement Approaches that try to find the best location in a network to deploy a VNF or VNF Chain.

Chaining How to design efficient VNF chains[1] and ensure policy enforcement in presence of non-passive (packet modifying) VNFs.

Consolidation Consolidation approaches investigate possible efficiency benefits from merging different VNFs to avoid redundant execution of the same processing steps

Frameworks VNF frameworks provide building blocks for implementing VNFs, differing in their focus on performance, usability and deployment model.

Orchestration The orchestration required to manage the lifecycle and scaling of VNFs and coordinated changes to the network

State Management State management approaches are concerned with ways to synchronize state between VNF instances

Hardware Acceleration There are several approaches to utilize different hardware solutions for VNF implementation

Application Use Cases Explore the possibility to leverage VNF workflows to improve application performance

This categorization is of course not exclusive, many approaches cover multiple of these aspects and some of the categories are closely related. This categorizations does however already show that there are many complex problems to solve and consider, often with impact on other aspects and dependencies on external factors.

[1] while most literature uses the term chain, the actual topology in most cases is some DAG.

3.1 Placement

Several approaches[2], for example [ABFL15; CLNR15; PLH+15; SER+12] or [GGA+12], cover the NFV placement problem, i.e. where to place NFs while optimizing resource usage (link bandwidth, compute resources, . . .) with latency (chain hops), and scaling potential. Most solutions assume the potential to run multiple VNF instances on the same host, which has its own specific challenges. The sections on Consolidation (3.3) and Frameworks (3.4) will discuss these in more detail. Finding an optimal solution for the placement problem is NP-hard [LQ16], even when only considering standalone NFs. Chains (DAGs) of NFs, which are often required, further complicate this problem. A popular solution (e.g. [CLNR15]) is to formalize the placement problem as an Integer Linear Programming (ILP) model.

E2 [PLH+15] uses a different approach. An initial valid (complies with resource constraints) placement of VNFs on hosts is found using a depth-first search. Then, until no further improvement can be made, the pair of VNFs on different hosts that would lead to the largest reduction in cross host traffic when swapped, are swapped.

Slick [ABFL15], among other criteria, uses the inflation factor (IF), defined as $log(f_{out}/f_{in})$ where f_{out} and f_{in} are the output and input traffic, of a NF to determine its placement closer to the source (negative IF), or sink (positive IF) of a traffic flow.

Stratos [GGA+12] optimizes the placement to minimize the load on inter-rack links in data centers, which are often oversubscribed and can become a bottleneck. Stratos tries to place all VNF instances in the same rack while also incorporating the potential need to scale the instances during runtime.

However, the extend to which the placement problem actually exists in real world use cases, is not clear. It seems realistic to assume that most VNF deployments may want to use specialized hardware (cf. 3.7), limiting the number of possible VNF hosts. Network administrators might also prefer the keep the forwarding logic of the network simple, limiting service chains entry and exit points to switches that are directly (and possibly exclusively) connected to VNF hosts, further limiting the problem space to a point where the placement problem becomes trivial to solve.

3.2 Chaining

Most network functions are not deployed in isolation but are part of a network function chain [3]. These chains are defined based on policies that define what traffic must traverse what functions in what order, with possible branching after each function. In conventional networks the chaining is mostly enforced by the physical network topology. In environments where the network and compute resources are virtualized and highly flexible, the enforcement of these policies has to adapt to changing topology, work with distributed network functions, deal with complex forwarding rules while optimizing bandwidth and resource utilization and ensure correctness, i.e. avoidance of loops.

[2] see [LQ16] for a survey

[3] often also called service chain.

SIMPLE [QTC+13] is a system that can translate high level chaining policies, for example external traffic to the web server has to pass through the firewall and IDS, to OpenFlow rules. It considers resource constraints for the rule generation and requires no changes to network functions. If the input port of a packet is not sufficient to determine the processing status of the packet, SIMPLE encodes state in header fields to ensure correct forwarding. One major problem are mangling (packet modifying) network functions, for example NAT that might expose unpredictable output flows. To cope with such NFs in a chain, SIMPLE proposes to use payload correlation to determine the mapping of flows before and after a mangling NF.

The authors of FlowTags [FCS+14], which is partly based on SIMPLE, argue that when two properties, OriginBinding (the original identity of a packet should be used as identifier) and PathsFollowPolicy (ensuring that a packet is processed by all NFs in the correct order in a chain it is subject to), can be enforced in a NFV system, would enable to deal with mangling NFs in a more elegant way and to define policies in a more abstract way. FlowTags proposes to reactively mark new arriving flows with an unique tag (can be per-flow or more course-grained, e.g. source subnet) that can be used to determine its original identity, even through mangling NFs. Middleboxes can also encode metadata in a tag, for example an IDS might classify a packet as dangerous, allowing to specify conditional processing steps. In order to allow NFs to restore the original packet identity and to encode processing results in the tags, NFs have to be modified, in contrast to SIMPLE. Similar to SIMPLE, FlowTags are encoded in VLAN headers or other fields that are unused by the rest of the network.

3.3 Consolidation

Several approaches explore the possibility to run multiple NFs on the same host, most commonly as VMs or containers, while some run in a single executable (cf. section 3.4). CoMb [SER+12] and OpenBox [BHH16] push this further and consolidate different NFs on a sub-process level.

CoMb [SER+12] is one of the earliest works on NFV. CoMb describes a consolidation architecture based on the Click [KMC+00] software router. CoMb offers reusable building blocks for session management, TCP stream reconstruction or higher level protocol parser, that NFs can depend upon, for example a HTTPS load balancer application will use the session and TCP/HTTP reconstruction modules. For each traffic class CoMB will merge the series of modules (defined by a policy) and their dependencies into a so called hyperapp, that will be pined to a core and uses a run-to-completion model.

OpenBox [BHH16] offers a similar set of abstract processing modules that NFs can use. Compared to CoMb, OpenBox further increases the reuse of modules. OpenBox describes a NF as a processing graph (DAG), consisting of the different processing steps. OpenBox will merge the graphs of different NFs, for example a firewall and IPS, that are running on the same host, with a novel merging algorithm that optimizes for the path length while ensuring that the operations are performed in the same order as passing a packet through both graphs consecutively. OpenBox can also merge common steps like header classification, the merged step will produce the correct union of both merged steps. OpenBox also allows to pass metadata, like the classification, between processing boxes, using NSH (network service headers), which allows to perform some processing steps on different network elements. For example switch ASIC could be used for the classification.

3.4 Frameworks

Implementing a VNF is a complex endeavor. A VNF should achieve high performance (throughput/latency) while being resource efficient and easy to maintain and update. However, the low level nature of many commonly used VNF building blocks like DPDK make VNF development slow and error prone. It is also common to execute multiple NFs on the same host, reducing the reduction of cross VNF host bandwidth utilization, chain latency and increasing resource efficiency. In this case, the correct and performant chaining and strict isolation between local NFs are additional complex problems to solve. Several frameworks for VNF/VNF on-host chain implementation exist. They offer a higher level abstraction for packet processing and chain definition, reducing the work required to implement common tasks like packet IO, allowing to focus on implementing the actual functionality of a NF. Implementation frameworks can be classified into VM/container based approaches and single process solutions, based on the format that is used to define NFs and on their data abstraction, i.e. packets or byte stream[4].

The dataplane of E2 [PLH+15], called E2D, is based on SoftNIC (also called BESS) [HJP+15], a modular software router similar to Click that is based on DPDK. NFs are defined as SoftNIC modules and can leverage an additional E2 API that provides zero-copy packet transfer and the possibility to transfer per-packet metadata, as well as signaling between NFs. It is also possible for E2 modules to operate on a TCP byte stream instead of discrete packets, increasing the efficiency of a chain if multiple NFs operate on the byte stream abstraction as the stream only has to be reconstructed once. E2 uses a run-to-completion model and batching to minimize per-packet overhead.

In OpenBox [BHH16], a NF is defined as a control plane application that contains a definition of the processing modules in the OpenBox data plane it uses and how they are interconnected. To implement a new NF, the existing catalog of over 40 modules can be used or new modules can be defined. OpenBox Modules are implemented as Click modules. Modules can send events to the control application and the application can request statistics or modify module state. As already mentioned in 3.3, OpenBox allows to pass metadata between NFs.

Similar to OpenBox, Slick [ABFL15] allows to define high level applications in Python using a simple API that controls and reacts to events from data plane Click elements, that perform the actual processing.

NetBricks [PHJ+16] uses the Rust programming language to provide isolation between different NFs and a high level pipeline model for packet processing based on DPDK. A NF in NetBricks is implemented as pipeline of high level collection operations like parsing, filtering, modifying, group by, shuffle and merge, that are using closures provided by the NF developer. Pipelines are pinned to a particular core and read and write from a specific NIC port. However, it is also possible to define more complex processing topologies and core assignments using group by, shuffle and merge operations. NetBricks also offers a byte stream abstraction for NFs operating on TCP streams and state synchronization capabilities between cores. NF chains are defined at compile time in code, offering a less flexible solution compared to other frameworks. This however allows the compiler to optimize the combined program. The Rust compiler also ensures memory isolation between different network functions. Once a pipeline step has processed a packet, the compiler guarantees

[4] Reconstructed TCP data stream

that it can not modify it at a later point[5]. In other approaches this is enforced by copying the packet or not addressed at all. NetBricks uses batching and a run-to-completion model. Packets are not copied between NFs, the pipelines of different NFs are just invoked consecutively.

mOS [JMK+17] is a framework for writing NFs, primitively targeting TCP byte stream based monitoring use cases. It is based on mTCP [JWJ+14], a high performance user space TCP stack. NFs in mOs are written in C and use an event/callback based API provided by mOS. The traffic can be filtered based on the flows 5-tuple and events include for example packet in, retransmits, out-of-order or connection start and end. mOS does not directly address NF chaining. Batching and a run-to-completion model are used, and flows are distributed between cores by symmetric RSS.

OpenNetVM [ZLZ+16] uses containers as NF format. Shared memory is used to store packets which are received via DPDK. It only copies packet descriptions (memory address, metadata) between NF containers, enabling zero-copy processing. Each NF container is run on a dedicated core and uses an OpenNetVM specific library to poll for new packets, which are then processed in batches. Chains can be defined based on a flow table or NF output. They can be changed during runtime and OpenNetVM is able to load balance traffic between instances of the same NF via explicit flow rules.

Flurries [ZHR+16], which is based on OpenNetVM, extends the OpenNetVM model from a single NF per core busy-polling, to multiple NFs per core that are waiting to be activated and a single container for each NF/flow pair. It can run hundredths of NF containers that will only process packets of a single flow. The Flurries manager will wake up NF containers (using semaphores) once enough packets are in its RX queue to reach the batching threshold or if a maximum wait time has been reached. The NF will then process the packet batch and wait for the next batch, in the meantime the manager might wake up another NF running on the same core. Each NF can also be assigned a priority to influence their wake up frequency, which can be useful for new TCP flows during connection establishment. To minimize startup latency for new flows, Flurries maintains a pool of idle NF containers and will reuse containers of terminated NFs after resetting their internal state.

xOMB [ABK+12], published in 2012, describes a NF architecture geared towards request/reply oriented load balancing workloads on top of TCP. NFs are written in C++ as pipeline modules that are executed in order until a module stops the execution chain for a request. Modules can be synchronous or asynchronous, i.e. block processing until a response has been received. Modules can append metadata to requests and responses and share state. xOMB will ensure that replies are sent to a client in the order that the requests were received, even if the back end connections are responding in a different order.

Flick [ACM+16] is an approach that uses a DSL (domain specific language) to describe application specific NFs operating on TCP. Flick programs are compiled to C++ and executed on (linked to) the Flick runtime, which is based on mTCP [JWJ+14] and DPDK. Each incoming TCP connection is assigned its own task graph, i.e. instance of the compiled Flick program, for example parsing, processing and serializing of a protocol message are separate tasks. Each task is assigned to a fixed core, allowing the different processing steps to be performed on different cores, which can be useful for compute bound tasks. However, most other approaches choose locality and a run-to-completion model over increased parallelism. Each task has its own input queue and once input is available,

[5] Rust ensures that at any point only one mutable reference to a memory location can exist

gets placed in a FIFO queue of a worker thread that is assigned to a fixed core. Worker threads will execute their queue or try to steal work from other workers if their queue is empty. Workers will run a task for a fixed duration, which is ensured by the compiler, or until it has no further input. The DSL allows to define protocol messages and supports dynamically sized fields and is limited in its expressiveness, for example Flick programs can't contain unbound loops and in consequence can't starve other programs.

ClickOS [MAR+14] implements a VNF platform based on the Xen hypervisor and the Click [KMC+00] modular software router. NFs are implemented as Click modules and run inside a VM that runs a custom, barebone OS that directly executes the NF Click module. The Xen domain VM runs a custom software switch (for performance reasons) based on VALE that is connected to the hosts NIC ports and the NF VMs.

3.5 Orchestration

NFV orchestration frameworks involve many of the previously discussed concepts. They must be able to compose NF chains spanning multiple VNF hosts, constantly adapt to policy and load changes, while trying to optimize the overall resource utilization and performance characteristics as well as ensuring that the network is updated in synchronization. The lines between implementation and orchestration framework are somewhat blurry, many of the implementation framework handle orchestration tasks on a single host. Therefore we limit the discussion to multi-host orchestration.

E2 [PLH+15] is the most complete solution for NFV orchestration. It covers placement, chain composition as well as scaling. While E2 tries to avoid performance bottlenecks and unnecessary inter-host communication during the initial VNF placement (cf. 3.1), it uses estimates of the performance characteristics of a VNF to calculate the instance count and placement. During runtime E2 can react to performance regressions or load spikes by dynamically scaling VNF instances based on load statistics it collects from the E2D (cf. 3.4) and the VNF instances. A novel migration method is used during scale events to avoid state transfer between instances of the same VNF type, which is covered in more detail in section 3.6. In addition to the virtual switch implemented by E2D for in-host forwarding and filtering, E2 also manages a SDN switch that connects VNF hosts and chain ingress and egress ports.

Stratos [GGA+12], like E2, covers placement, chaining and scaling. Placement and scaling consider the network topology to avoid saturating oversubscribed inter-rack links (cf. 3.1). During scale events, which are triggered based on metrics reported by NF instances, Stratos will iterate over a NF chain and add, in order, instances of each NF until the reported metrics do not further improve. Downscale events are handled in a similar manner. The forwarding between different host is using a model called distributed programmable switch, which uses source routing to connect the NFs of a chain. A conventional IP routed fabric is used to transfer packets between hosts. The source host will lookup the target IP of the next chain element in a distributed forwarding table called FIB. To distribute load between instances of the same NF type in a chain, the per flow FIB entry is lazily decided on the first packet of a new flow based on the number of existing flows for each instance.

OpenBox [BHH16] also allows to scale (software based) data plane elements based on CPU load and memory usage, but does not go into detail on the implementation.

Apart from the approaches discussed, there are also several industry projects related to NFV management, for example OSM [osm18], ONAP [onap18] and OPNFV [opnfv18]. These projects mainly focus on telco use cases and utilize and integrate with related open source orchestration systems like OpenStack or OpenDaylight.

3.6 State Management

Most network functions have to maintain state to function. For example NAT stores the internal to external and reverse mappings of flows and free IPs/ports and a firewall must keep track of connection state. Other NFs that reconstruct the TCP byte stream have to maintain a buffer and keep track of the TCP protocol state. This stateful nature complicates distributing and dynamically scaling NFs as the state related to a packet is required to correctly handle that packet. Typical solutions that separate the state from the processing e.g. in elastic cloud architectures, are not suitable for NFs because of the very strict latency requirements and high packet rates. Different solutions have been proposed to partition, move or share state between NF instances and ensure that packets are forwarded to an instance that holds the state required to process it.

E2 [PLH+15] solves the state management problem by avoiding any state migration. If the E2 orchestration determines that a NF must be scaled up, E2 splits the flow space based on the flow hashes in half. This splitting however would redirect many flows that were handled by the existing instance. A simple solution would be to install higher priority rules for the flows that fall in the range assigned to the new instance but should be handled by the old instance. While the number of exceptions would shrink over time, initially the number might be very large and and exceed the capacity of hardware switches. E2 solves this by keeping the old forwarding rule on the hardware switch and perform the range-based rules and exception are installed on the VNF host of the old instance, i.e. packets to be handled by the new instance are forwarded to the VNF host of the old instance and then forwarded from there to the new instance/host. Once the number of exception rules falls under a threshold value, the rules are moved to the hardware switch. This scheme avoids excessive hardware switch rule counts at increased latency for flows assigned to the new instance and load on the VNF host of the old instance until the rules are moved to the hardware switch. For NFs that depend on state that is not scoped by flow, for example an IDS keeps state related to all flows involving a specific host, E2 allows to define how to partition the flow space for each NF. While this is a simple and elegant solution, it does not offer a way to implement fault tolerance and could potentially be susceptible to scaling problems with long lived flows.

OpenNF [GVP+14] argues that the possibility of moving state between VNF instances is a required feature of a NFV system. Different flows might have different load impact and simple solutions like used by E2 are not sufficient to handle such situations. The capability to move and share state also allows to perform maintenance tasks like updating instances or to implement fault tolerance. The authors further argue that drops and reordering of packets are typically no problem for endpoints, for NFs, depending on their properties, the move operation must guarantee that no packets are dropped or even reordered, otherwise the NF might expose incorrect behavior, for example an IDS might generate (false positive) warnings. OpenNF defines an API for NFs to implement (southbound) and the OpenNF controller exposes an API that NFs can use to trigger move or share operations (northbound). The southbound API provides methods that can import, export and delete state based on a scope and filter. The scope of a state is defined based on the flows it applies to, either a single

flow, a set of flows (e.g. related to a certain host) or all flows. The filter is a wildcardable match similar to an OpenFlow match. The API also defines methods that instruct the implementing NF to start and stop sending events for arriving packets based on a given filter to the controller. The instance will either process, drop or buffer packets that match the filter based on a given action. Apart from the loss and loss/reorder free move operations, OpenNF also provides a simple move operation. For a simple move the controller will first request the state that relates to the flow that should be moved from the old instance and delete it from there. Then the state will be installed on the new instance and switches, starting from the lowest common ancestor (LCA) switch of both instances, are reprogrammed to redirect the flow to the new instance. Any state update that will happen on the old instance after the state has been sent to the controller will be lost. For a loss free move the controller will instruct the old instance to send events for each received packet and drop the packet after sending the event before requesting the state. The controller will buffer the packets until the new instance signals that it applied the received state and inject them at the switch closest to the new instance and reprogram switch to forward to the new instance. Any event from the old instance received by the controller will be sent to the new instance and, after no new event has been received, which eventually will happen, the old instance will be instructed to remove the event trigger. To implement order preserving moves, OpenNF uses buffering at the controller and new instance, as well as a two phase forwarding update algorithm. Similar to the loss free move, packets arriving at the old instance will be buffered at the controller. After the new instance has applied the state, the controller will send the buffered packets received from the old instance to the new instance with a flag that indicates that these packets should be processed directly. The new instance will also be instructed to send an event for and buffer any packet that does not have the process directly flag. Next, the forwarding rule on the LCA switch for the moved flow is changed to forward to both, the old instance and the controller. After receiving the first packet from the switch, the controller will install a new rule with higher priority on the switch that forwards to the new instance, but at the same time keep track of the last packet it received from the switch. When the controller receives an event from the new instance that it processed the last packet the controller received from the switch (meaning all packets forwarded by the old instance have been processed), the controller will instruct the new instance to disable the buffering and process packets normally. In addition to the move operations, OpenNF also offers copy and share operations that allow state to be shared between instances. The copy operation offers eventual consistency by repetitively copying state. The share operation can offer either strict or strong (sequential) consistency, however at a high overhead. For strong consistency the NF instances will not process packets directly but send them to the controller which will serialize the events and send them back to the originating NF for processing. The controller will then request and forward the updated state to the other instances before handling the next packet. For strict consistency, all packets (of the flows that are configured for strict consistency) are forwarded to the controller instead of NFs and perform the same state update as in strong consistency. In [GA15] two improvements to the original OpenNF system are proposed. The first improvement is packet reprocessing. During a move operation, the old instance will continue to process packets and, if a packet caused a state update, will forward a copy to the new instance that will also process the packet, hence receive the state update[6], but will drop the packet. The controller will buffer the copied packets until the new instance acknowledges that it applied the initial state transfer and then send the buffered packets marked with a drop flag. The second optimization is to cut the controller out of the state transfer and let the instances handle it

[6] Assuming deterministic NFs and that packets are not reordered

peer-to-peer. The major problem with the proposed move operations is the buffering of packets, which, at 10 Gigabit/s (or 1.25 Gigabyte/s), seems hard to realize, the state share operations come with even higher overhead.

StatelessNF [KCH+15] applies the concept of separating the application state from the processing, which is often found in elastic cloud environments, to network functions. StatelessNF utilizes RAMCloud, a distributed, low latency, in memory key-value store. NFs are running in containers on hosts that are connected to the RAMCloud instances via Infiniband. RAMCloud instances do not fully replicate their state in memory, for every stored entry there is only one node that keeps the entry in memory, other nodes will keep the entry on disk for fault tolerance. This can lead to misses and increased latency, if the NF instance will not query the correct RAMCloud instance. NFs use DPDK for packet IO and will process packets in multiple threads in parallel and batches. The interactions with the datastore (which takes about 6µs for a read and 15 µs for a write) are further batched over all threads, amortizing the remote data access overhead. The RAMCloud interface is extended to support timers, simplifying the interactions with the datastore. StatenessNF utilizes a SDN controller and a switch to distribute traffic between hosts and instances.

Split/Merge[7] [RWJW13] is an approach that, similar to OpenNF, identifies that VNF state is often tied to a specific flow (or shared between instances) and leverage that insight to scale out (split) and scale in (merge) VNF instances. Similar to OpenNF, an API/library is offered to VNFs that implements the state move logic. In contrast to OpenNF, where the VNF code *owns* the state, in Split/Merge the state is hold by the library and to access flow state the application code has to retrieve the state and once finished processing a packet of that flow, return the flow state to the library. Split/Merge will wait until a state is not in usage if the particular flow should be moved to another instance. Once the state has been moved, the library will return an error, indicating that the packet should be dropped, if further packets of that flow are accessed on the old instance. During a move the flows pointing to the old instance are removed and packets are buffered at the controller until the new instance is ready to process the flow. When the new instance is ready, the controller will install new forwarding rules to the new instance and inject the buffered packets at the switch closest to the new instance. Similar to the discussion in the context of OpenNF, this move operation is not drop free, as the old instance might drop packets received after the flow was moved and before the network update finished. The move might also reorder packets as there is a race condition between the controller injecting buffered packets and the network update allowing packets to reach the new instance.

Pico Replication [RWJ13] extends Split/Merge to provide fault tolerance. In addition to the state management API that Split/Merge provides, Pico Replication adds an API for packet IO. A NF will receive packets from the packet IO API of Pico, access and potentially modify the state for that flow, process the packet and hand it back to the packet IO API of Pico. Pico will periodically copy flow state to another instance. During such a snapshot operation the packet IO API will buffer all packets belonging to the flow that is copied and only pass them to the calling NF once the snapshot operation finishes. Similarly the packet IO layer will buffer outgoing packets of a flow until a snapshot operation completed. This guarantees that any packet that (potentially) impacted the state will not be released to the network until the state has been replicated to another instance. The snapshot interval can be configured for each flow individually, going as low as 1ms. For NFs that

[7] Split/Merge is the name of the paper and concept, the implementation is called FreeFlow

only update their state once per flow, the replication can also be disabled after the initial replication. If an instance fails, all active flows on that instance will be rerouted to an instance that holds the backup state for that flow.

FTMB [SGB+15] proposes a fault tolerance solution for NFs. The approach is based on well known rollback-recovery approaches from distributed systems literature, but uses two new techniques that are suited for the requirements of a NFV use case called ordered logging and parallel release. Sherry et al. make two observations that motivate these new techniques. First, a simple replay based recovery, where the logged inputs since the last state snapshot are replayed at the backup instance in a failover event is not sufficient to correctly recover state as NFs are highly parallel, rendering the processing of packets that access non-flow scoped state non-deterministic. Second, the latency overhead introduced by blocking the forwarding of packets until the state has been copied to a backup location (as in Pico), is too high. To solve the first problem via ordered logging, FTMB records a packet access log (PAL) for each state access that happens during the processing of a packet. A PAL is a tuple $(p_i, n_{ij}, v_j, s_{ij})$, where p_i is the packet, v_j the accessed state (variable) and n_{ij} counts the total number of state accesses so far for packet p_i and s_{ij} notes the total number of accesses to v_j. State variables are protected by a lock and the PAL for the access is placed in an output queue to the output logger before the lock is released, ensuring no reordering happens. For calls to non-deterministic operations like random number generation or time related methods, a PAL is created as well, but the returned value is stored instead of v_j and s_{ij}. With this information the backup instance can, during a failover event, reconstruct the exact same ordering of state accesses by controlling the order in which the locks protecting state variables are released and by using the stored return values non-deterministic calls. To solve the second problem, parallel release is used. FTMB assigns each PAL a per-thread sequence number before sending it to the output logger. When sending a packet to the output logger, FTMB will attach a vector clock of the current max sent PAL number of each thread to the packet. The output logger will keep track of the max received PAL sequence number of each processing thread and release a packet as soon as all vector elements of the packet are greater or equal to the max received PAL of each processing thread. To increase performance, packets can be batched so they share a vector clock, reducing the comparisons needed on the output logger. The system has no single point of failure, the master can continue to process packets even if the output logger crashes. To reduce input logger and PAL log size, periodic snapshots are taken and stored on the output logger.

3.7 Hardware Acceleration

Conventional network functions utilize specialized hardware, i.e. ASICs, to improve performance and energy efficiency. With the transition to NFV these benefits are given up in favor of higher flexibility by using typical server x64 hardware. In recent years there have been several approaches trying to utilize other hardware resources for packet processing, e.g. GPUs [GJM+17; HJPM11] or FPGAs [SGG+17]. These processing units lie somewhere in the middle ground between fully flexible but (comparatively) slow (CPU) and inflexible but fast (ASIC) [BRXM15].

[BRXM15] proposes an abstraction layer and orchestration (scheduling) workflow for VNFs to utilize different hardware accelerators, including GPUs, FPGAs, OpenFlow switches and smart NICs. One problem that remains is that network functions would have to be implemented for different processing platforms, as a VNF might have to run on CPU only if a hardware resource

could not be reserved. Having to implement the same VNF multiple times for each possible hardware target, would make it hard to develop and maintain such a system, probably voiding any advantage of VNF over traditional appliances. While this can be partly mitigated by emulating the hardware accelerators in the CPU, this has undesired impact on resource utilization, performance and in consequence capacity planning. This problem makes a strong point for a general network programming language that a VNF could be written in once and run on different hardware targets.

P4 [BDG+14] is such an attempt at a network programming language. It can target CPUs, FPGAs and specialized switch ASIC. Compared to OpenFlow, which only offers limited flexibility as the supported match fields and actions are defined in the protocol itself, P4 is more flexible: A P4 program can define new protocols and what the headers, tables and actions look like. However, P4 is no general language: It can only express functionality that fits in the typical match+action abstraction introduced by OpenFlow. Any functionality that does not fit this model must be implemented in the (slower) control plane and modify the tables in response.

Emu [SGG+17] is another DSL (domain specific language) for defining network functionality. Emu is based on C# and targets CPUs and FPGAs, claiming improved debuggability and more flexibility compared to P4.

ApuNET [GJM+17], among other apporaches [HJPM11; JHH+11], proposes a system to utilize GPUs, and specially APUs, for packet processing. For an IDS NF APUnet showed up to 4x the throughput of a CPU based solution. The overall system is however quite complex: It utilizes DPDK for zero-copy handling of the packets and deals with different cache and synchronization effects between CPU and GPU (or APU).

PBCE [BZ16] describes a technique that can be used to extend the often limited table capacity of OpenFlow hardware switches by directing a subset of traffic to an adjacent switch that has unused table capacity. If a switch's table capacity is getting close to being saturated, the PBCE control module will automatically install so called eviction rules that will forward traffic from a certain port to an adjacent switch and move all rules for that port to the same switch. The moved rules are rewritten to send traffic back to the original switch via the same port. The rewritten rules also include the port the original switch should output the rewritten packet as metadata, e.g. encoded via VLAN, MPLS, DSCP or other available fields. This way the original switch only has to install N rules, one eviction rule and N-1 backflow rules where N is the number of switch ports.

In [KDR17], Kohler et al. explore the possibility to run some parts of SDN control applications directly on switches. Since the control plane of modern white box switches is typically based on a specialized Linux distribution (e.g. ONL, ONOS, PicOS, Cumulus) that runs the default control applications (e.g. OpenFlow agents), running custom applications on the switch is easily possible, which could also be used in the context of NFV [KDBR17]. However, evaluation in [KDBR17] showed that the bandwidth between data and control plane is limited, reducing the potential for NFV use-cases where the control plane has to frequently handle data plane events.

3.8 Application Use Cases

NetAgg [MRA+14], DAIET [SAA+17] and P4CEP [KMD+18] explore the possibility to use middleboxes and/or programmable forwarding elements to perform certain application workloads. NetAgg runs the aggregate phase of a partition/aggregation workload (e.g. map/reduce) on boxes

that lie on the logical aggregation path, reducing network congestion. DAIET applies the same concept to machine learning applications, but performs the aggregation workloads on programmable forwarding elements, i.e. a P4 capable switch. P4CEP describes how operations in complex event processing (CEP) systems can be performed on-path in networking elements using the abstractions that P4 offers.

While NetAgg and DAIET report large reductions in network load, they break the assumption that applications are independent from the network architecture, introducing possible conflicts with cloud-like operations.

4 System and Entity Model

In this chapter, the system model that is used for the remainder of this thesis, is presented. It is based on the system models found in literature but differs in some aspects, i.e. the inclusion of the control plane of hardware switches or assumptions about the deployment environment. The following sections will discuss the system model and entity models of the network, network functions, chain definitions and orchestration respectively.

The overall system model is depicted in Figure 4.1. Packets from an adjacent network are passed into the NFV system via designated gateway ports. OpenFlow capable switches, NF offloading (NFO) capable switches and NF servers are connected via network links, forming the production network. Network function instances can run on the NF servers or the NFO switches, where the NF functions on the switch are running in the control plane of the switch but perform the NF in the data plane using an abstraction like OpenFlow or P4 to program the data-plane. Due to the results of [KDBR17] and our own evaluation, the system model only considers data plane capable network functions. Network functions are assigned a (virtual) port pair or a single port depending on the bidirectional traffic requirement of the NF. While NFs can be implemented multihreaded

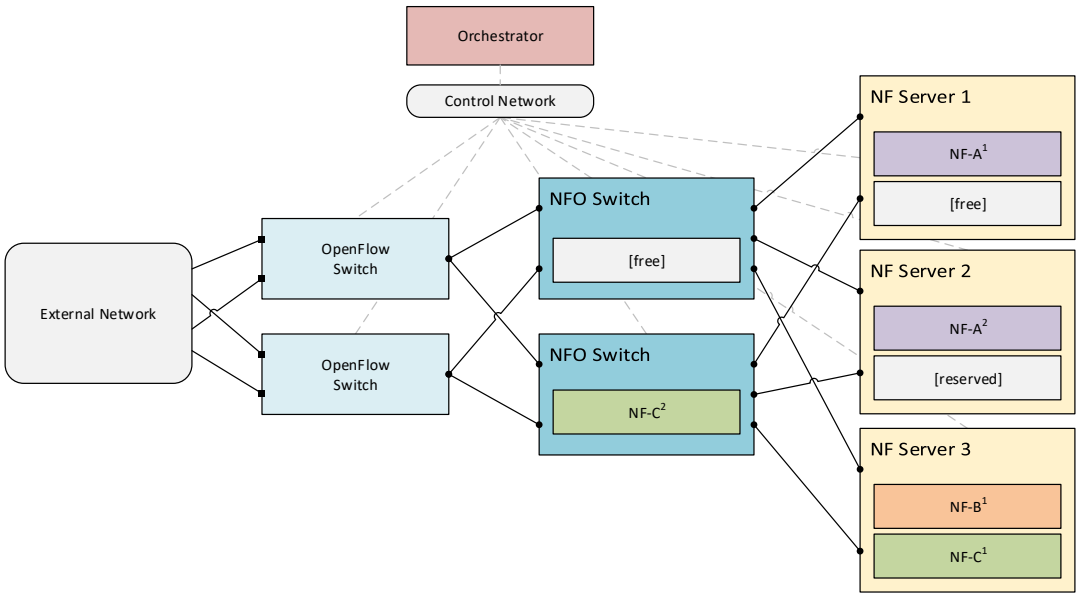


Figure 4.1: System model of the proposed design

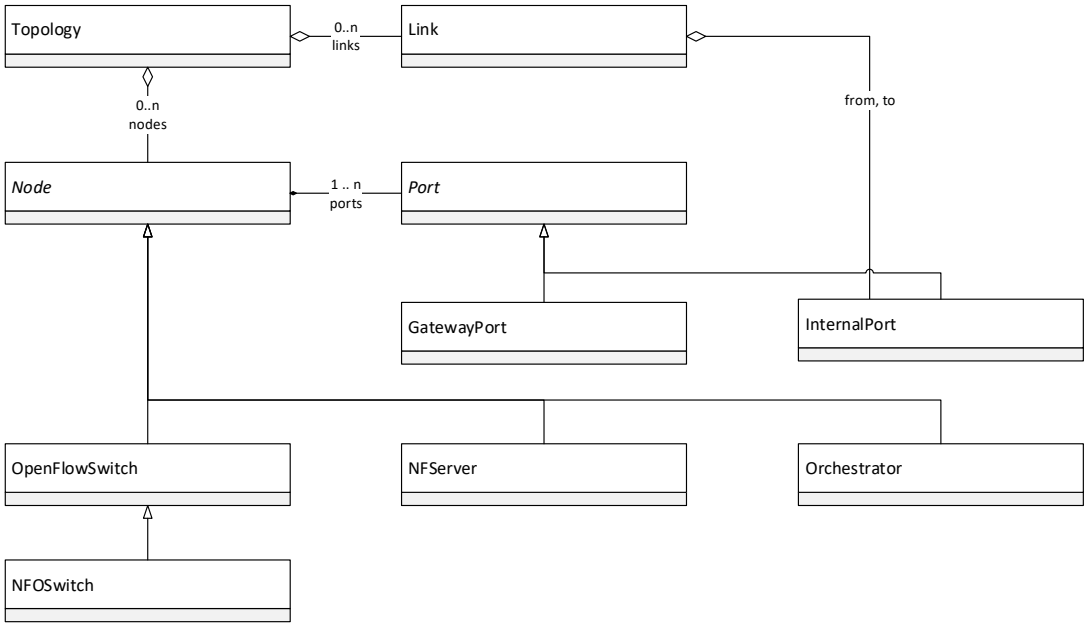


Figure 4.2: Network entity model

and generally are, the required steering or state synchronization inside a single NF instance is not part of the system model, but could be modeled as separate instances. Only fail-stop failures are assumed. A separate control network connects the switches and servers with an orchestrator.

4.1 Network Entities

Out of practical consideration, it is assumed that NFV workloads will run on a dedicated, non-distributed subset of a data center, e.g. a single rack or adjacent racks, with clearly defined borders between the NFV system and the rest of the network. Figure 4.2 shows the abstract model of entities that are part of the system model. A topology is defined by abstract nodes that have ports which are connected via a link. Ports can be either a GatewayPort or an InternalPort, distinguishing the connections to the external network and ports that connect system nodes. A node can either be a switch, a NFServer or the orchestrator. Two different types of switches are considered: Physical or virtual OpenFlow capable switches and physical switches that can additionally run network functions in their control plane. Note that this model does not explicitly consider virtual switches running on the NFServers that could directly connect NF instances running on a server. This reduces the complexity of the model without reducing the expressiveness, if needed virtual switches and ports can still be represented. In addition to the production network, a distinct, non programmable control network that connects all switches, NFServers and the orchestrator is present and is used for control, state and event messages. All node entities report their resource utilization, e.g. CPU, memory and bandwidth to the orchestrator.

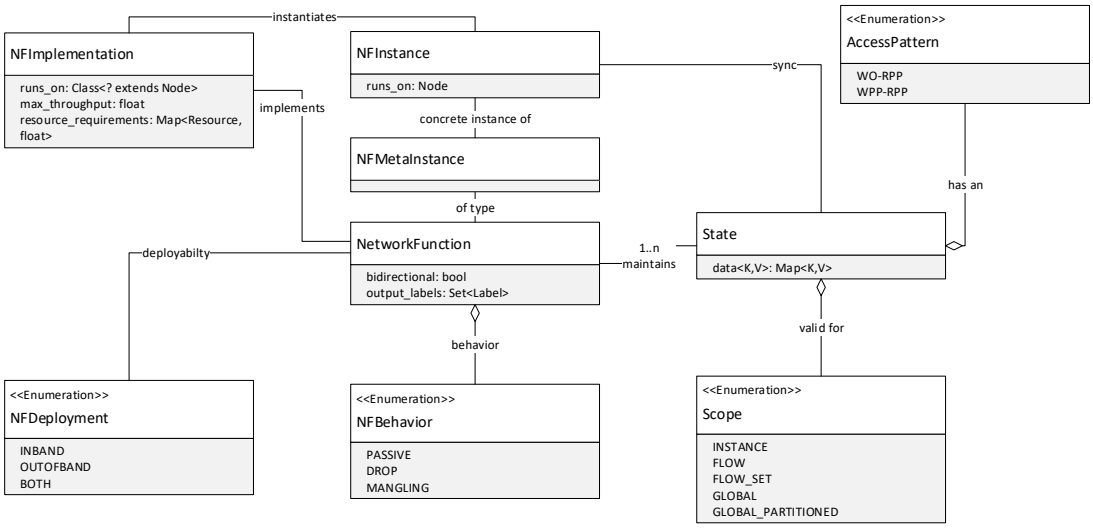


Figure 4.3: Network function model

4.2 Network Function

Figure 4.3 shows the model of a network function. A network function has high level properties like a deployment model (can work with a copy of the traffic or has to process the live traffic), behavior (will modify the traffic passing though it) or if it needs to see bidirectional traffic. Labels that the network function might produce are also defined. A network function might also have state, which is classified by its scope relative to the processed packets and the access pattern of that state. Each network function can have multiple implementations for different execution nodes, e.g. a NFServer or a NFOSwitch. The implementations define the resource requirements and their expected performance. A implementation might be instantiated as a NFMetaInstance. A NFMetaInstance in turn can be instantiated as a NFInstance. This indirection allows to express independent instances of the same type or inter chain dependencies on the same NF via the NFMetaInstance, without requiring to link via a concrete instance.

4.3 Network Function Chain

Figure 4.4 shows how a high level network function chain is modeled. A chain has a fixed starting point at a GatewayPort, a bandwidth requirement as well as scaling and fault tolerance policies. The scaling policy defines if the network functions in the chain should be scaled dynamically, based on the current traffic, or if all instances required to meet the throughput requirement should always be instanced. The fault tolerance policy, among other factors, is used to determine how instances share or partition state and traffic flows. Each chain has a single root processing step. A processing step is defined as a set of non-overlapping filters and associated actions that are executed if the filter matches. A filter may be a flow match or a tag match. An action can be to either drop the packet, output it to the network on a GatewayPort or to send it to a network function. The ApplyFunction action also defines how the output labels of the network function should be mapped to a flow tag

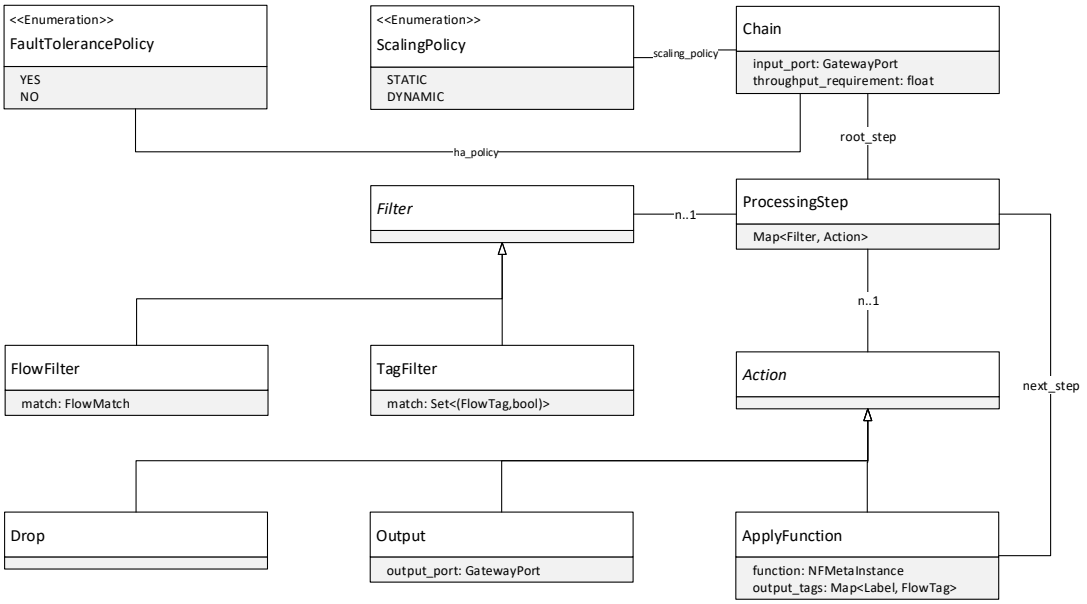


Figure 4.4: Network function chain model

and what processing step to execute after the function has been applied. A single NFMetaInstance can be used in ApplyFunctions of different chains, allowing to express that a NF will process traffic from different chains, which can be used to handle bidirectional flows.

While there are multiple network functions that work on higher level abstractions, e.g. an IDS will reconstruct the TCP stream and scan it for known malware signatures. Several approaches in the literature (e.g. E2 [PLH+15], mOS [JMK+17]) provide a way for network functions to directly work on this abstraction and to avoid the overhead of reconstruction this state at several NFs. We consciously exclude such an abstraction for the sake of simplicity and practicability.

4.4 Orchestration

In Figure 4.5 the orchestration entities are shown. The orchestrator receives the user defined chains, a topology graph and other configuration as input. Based on the input the orchestrator will compute a mapping based on some placement algorithm and the high level chain definition entities to filter and action instances. The filter as well as the drop and output action instances are mapped to one or more forwarding rules on switches and the ApplyActions are mapped to one or more NFInstances which will run on either a NFOSwitch or a NFServer. A ElasticyPlan is also part of the computed mapping, defining where backup NFInstances should be started, if necessary, and what forwarding rules must be installed to forward traffic to the new instances, reducing the reaction time for scale events compared to incremental placement computation. The orchestrator further acts as (or talks to) a SDN controller to configure switches. Network functions running on a NFOSwitch may also program the switch they run on.

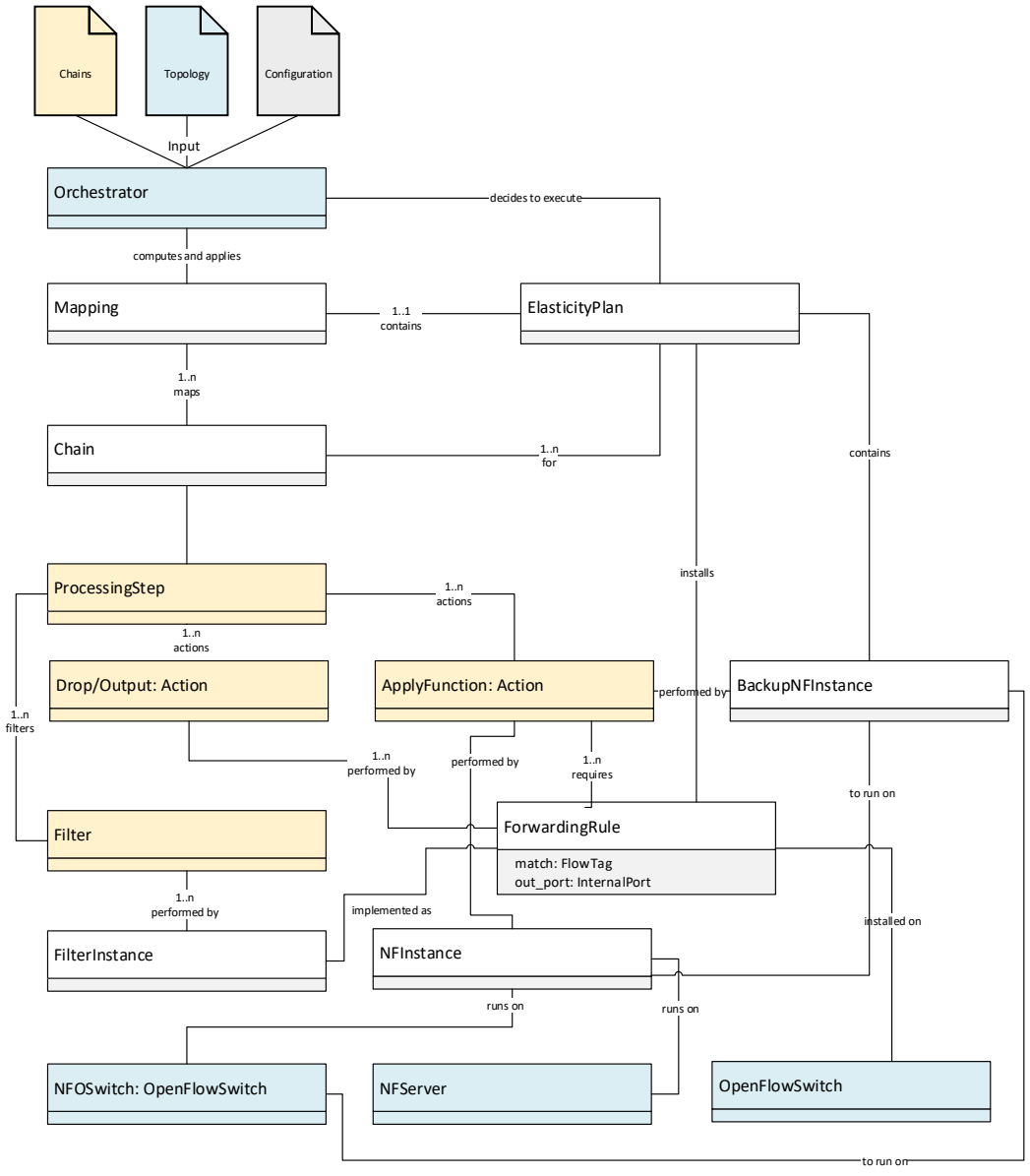


Figure 4.5: Orchestration model

5 Design

Based on the related work and presented system model, this chapter will discuss design decisions related to the system model and patterns presented in the previous/next chapter. While all of the topics discussed under related work are relevant for NFV systems, the core concern of this thesis, the distribution of network functions, is mostly concerned with two of these topics. First, dealing with network function state and second, the correct forwarding of packets to the different instances according to chain rules, state affinity and capacity constraints. Many of the systems presented in literature are very complex and/or have a high overhead/limited practicability, so choosing the correct pattern based on given variables can greatly impact the performance and other properties of the resulting system.

The major constraint, that has to be ensured to hold at all times, is that a packet is only processed by an instance that holds the state that is associated with the packet, otherwise the correctness of a network function can be violated. For example a NAT with two instances that has state for a flow on one instance but not on the other, will create a seconds mapping if packets of that flow are processed by the other instance and break the communication between endpoints. The two extreme solutions are to either maintain the state on all instances, allowing packets to be processed by any instance, or to only keep state on a single instance, requiring that all packets are forwarded to the correct instance.

The following sections will discuss the solution space based on existing literature, describe what pattern to use and presents the benefits switch offloading can offer.

5.1 Network Function Properties

The state that network functions maintain, can be classified based on four properties: Type, scope, consistency and access pattern, as shown in Table 5.1. This classification is based on results from literature, i.e. [KCH+15; RWJ13] and [GVP+14]. The state type differentiates state that was introduced by the system itself or an administrator and state that is created as reaction to some data plane event, which is more interesting as configuration state is typically read-only. The scope defines the context and/or duration that a scope is valid for. Instance scoped state is only required on a particular NF instance and doesn't have to be shared. Flow and Flow set scoped state is valid and indexed by a particular flow (5-tuple) or flow property (e.g. source or destination host) respectively and must be available on all instances that process packets of the flow or flow set. The required consistency of state, that is shared between instances, has an impact on the complexity of the required replication solution. The access pattern describes how often state is modified. For some network functions (cf. Table 5.2) only the first packet of a flow will perform a write operation (create the state) to state that is related to that packet while the other packets will then only require a read on the created state.

State Dimensions	Values
Type	Configuration, Operational
Scope	Instance, Flow, Flow Set, Global
Access Pattern	Write-once / read per packet, write/read per packet
Consistency	Eventual, Serialized

Table 5.1: Network function state classification based on observations in [GVP+14; KCH+15; RWJ13]

NF	OSI Layers	State-Scope	State-AP	Deployment	Abstraction
Firewall	3,4	F	WO-RPP	I	Packet/Flow
Load Balancer	5,6,7	F/G	WO-RPP	I	Session
IPS/IDS	3-7	F/FS/G	WPP-RPP	I/O	Flow/Session
NAT	3 (, 4)	F/G	WO-RPP	I	Packet/Flow
Monitoring	3-7	F/FS/G	WPP-RPP	I/O	Packet/Flow/. . .
Cache	7	F/G	WPP-RPP	I	Session

Table 5.2: Properties of network functions based on observations in [GVP+14; KCH+15]

5.2 State Management

State management is trivial if there is only a single instance of a NF. However, often a single instance does not offer enough capacity to process all traffic, requiring the partition of traffic between instances. With more than one instance, the properties of NF state (see Table 5.1) have to be considered so that the instance affinity property holds. When the NF in question only maintains state with a single scope, e.g. flow or destination host, a non-overlapping partition is possible and using this scope to partition the traffic will ensure that the instance affinity property is enforced. While this single-scope state property applies to a wide range of NFs, others maintain heterogeneous state with different scopes. When the scopes of the different sub-states overlap (e.g. state that is scoped by the source host and state that concerns to all UDP flows) or require a global scope, an exclusive partitioning is not possible, requiring that this state is replicated on multiple instances. This is also the case if fault tolerance is required, but for fault tolerance the complete state has to be replicated. Additional requirements, like load balancing might require additional capabilities, e.g. moving state between instances.

By looking at the defined properties of NF state, functional requirements and fault tolerance/scaling policies as well as the available resources/environment, it is possible to decide on a state management approach that fits best for the concrete use case. Based on that, Figure 5.1 shows a flowchart that can be used as a starting point to decide on a state management pattern.

The external datastore pattern, where state is kept externally from the NF instances, can be applied to most state management requirements, but might not be always realizable based on execution environment. The fault tolerance requirement can be used as a general discriminator. If fault tolerance is not required, simpler patterns can be used. If no overlapping state scopes have to be considered and no fault tolerance, partitioning is sufficient. If active load balancing is required, a move pattern has to chosen depending on the required semantics. If states overlap, a share pattern

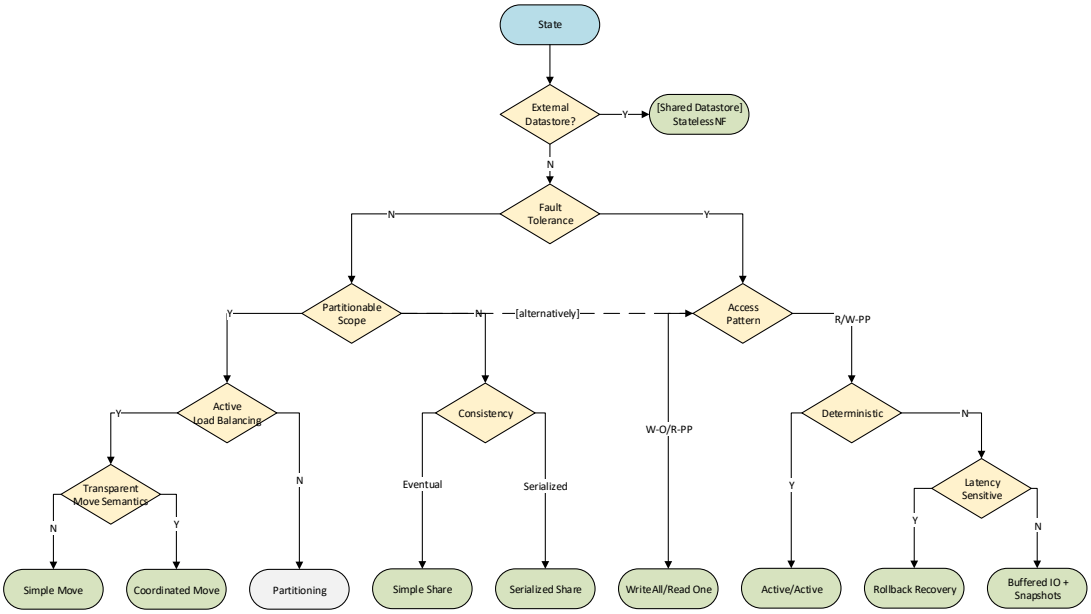


Figure 5.1: Decision tree for a state management pattern

has to be chosen, again depending on the required consistency. Alternatively the fault tolerance patterns can also deal with shared state. The simplest pattern that can provide fault tolerance is the write all / read any pattern. If, depending on the access pattern the write all / read any pattern cannot be used, the active/active, rollback recovery and buffered IO + snapshots patterns can be used.

Fault tolerant setups also have to comply to the output commit property, stating that a packet may only be released to the network after associated state has been synchronized to another instance [SGB+15]. This property ensures that after a failover event no state is lost that has modified the system external state, but typically introduces a latency penalty.

It is also notable that in the proposed system model the orchestrator is not involved in the state synchronization process. It will only create instances, pass them their configuration containing addresses of other instances, the configured scaling and fault tolerance policies and maybe partitioned configuration state (e.g. NAT pool). All operational state synchronization happens directly between the instances, reducing the load on the orchestrator and offering more flexibility to the implementation.

Certain patterns e.g. the migration avoidance scheme of E2 require virtual switches to run on the NF servers. The presented system design does not explicitly consider virtual switches with special capabilities, the functionality that the migration avoidance (and fallthrough) pattern requires could also be implemented in the application, for example via a reusable library.

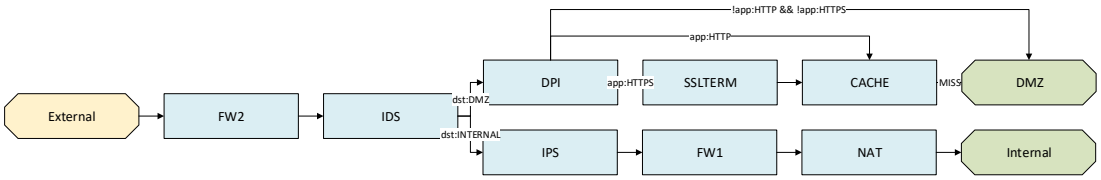


Figure 5.2: Example chain

5.3 Chains, Scaling and Steering

As mentioned in the context of state management, the instance affinity property, meaning that packets must be processed on an instance that has the state associated with the packet, is also a major concern for the forwarding part of a NFV system. An additional property that must be followed is the correct chaining, i.e. packets must be processed by the network functions defined in a chain, in the correct order and with the correct branching.

These two properties can be considered as two distinct forwarding/steering problems to solve. There is the forwarding in a chain from one processing step to another (the macro forwarding) and the micro forwarding to a concrete instance of some meta-instance that must maintain instance affinity and support load-balancing/scaling behavior. The hand-off from macro to micro forwarding could for example happen on the lowest common ancestor switch of all instances or some other location that was computed along with the placement.

For macro forwarding, the system model assumes to use tags (based on FlowTags [FCS+14]) that indicate the abstract source and destination of a packet, for example internal host to server in the DMZ as well as the process of the packet in a chain (has already been processed by the firewall) and optional metadata that instances can attach to a packet (e.g. a deep packet inspection NF can classify a packet by application) which can be used to conditionally decide the next processing step in a chain. The initial classification can happen at the gateway switches based on user provided match rules. The gateway switches will also remove the flow tag again if a packet leaves the NFV system. As already mentioned, technically flow tags can be implemented by using existing header fields like the VLAN tag, MPLS or specialized protocols like NSH. The orchestrator can, based on the given chains and tags, compute the assignment of all valid (the Cartesian product of all tag dimensions (e.g. src, dst, proc and other metadata like L7 application or NF categorization) filtered for all combinations that are actually possible appear in a chain) tags to an actual VLAN tag or MPLS label(s). Figure 5.2 shows an example chain that defines the processing for inbound traffic to a typical enterprise network. All traffic must first pass through a firewall and an IDS. After the IDS, the traffic is split based on a destination tag. Internal traffic will be processed by an IPS, another firewall and a NAT before being sent to the internal network. Traffic to the DMZ servers is sent to an DPI (deep packet inspection) NF to determine and subsequently split the traffic based on application protocol. HTTP traffic is sent to an cache and, in the case of an cache miss, to the DMZ network. HTTPS traffic is first sent to an SSL endpoint before being sent to the cache while non HTTP/HTTPS traffic is sent directly to the DMZ network. A HTTP packet would be tagged as src:INTERNAL, dst:DMZ and app:HTTP. An additional tag would indicate the processing status so that a switch that sees the packet multiple times can determine where to send it next.

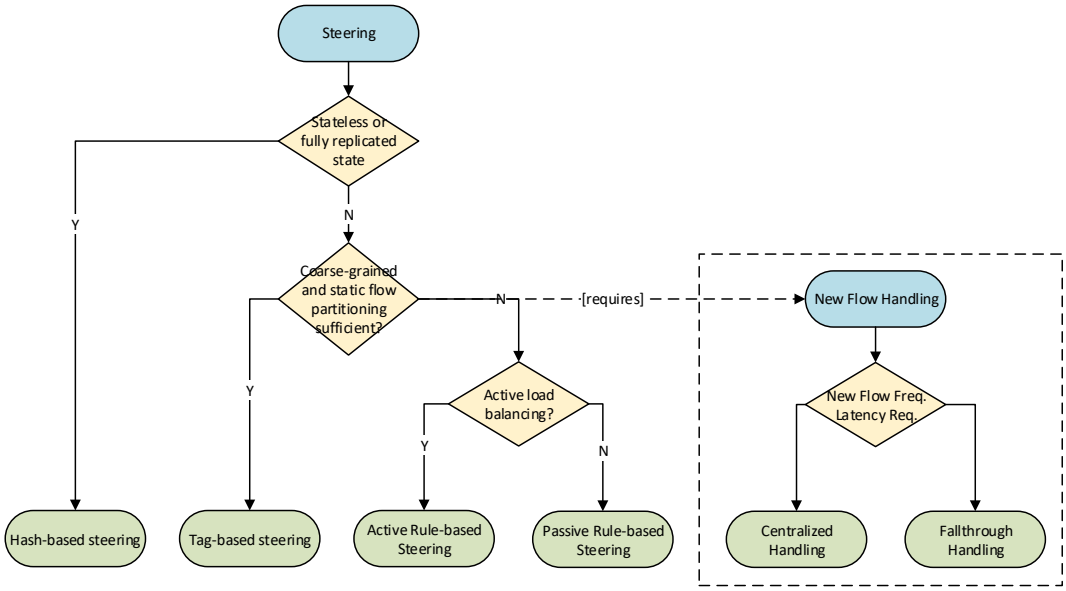


Figure 5.3: Decision tree for a steering pattern

The most obvious solutions for micro forwarding are to either use a hash based approach or to use explicit rules. The hash based solution is obviously simpler and there are hashing solutions that can deal with bidirectional traffic (symmetric hashing) and, to some degree, changing instance sets (resilient hashing, e.g. consistent hashing [KLL+97] or rendezvous hashing [TR98])[1]. While resilient hashing solutions ensure that the mapping remains stable when an instance leaves, meaning that only flows that were hashed to the leaving instance are relocated, for a joining node a random set of flows will be remapped. This limits hash based solutions to cases where there is no state, the state is available on every instance (e.g. shared datastore), the set of instances is stable. The rule based solution uses flow rules to specify what instance will process what flow. While this stateful solution supports scaling, fault tolerance and load balancing, the solution itself does not scale that well. The flow tables of switches are limited in their capacity and inserting a rule (or two in the bidirectional case) for each flow will saturate the tables quickly. It is possible to work around this capacity limit with a solution like PBCE [BZ16] where the tables or neighboring switches are used to extend the capacity of a flow. This however further increases the complexity of the system. There is a potential to save flow table space with more coarse grained rules (cf. the tag based forwarding discussed below), but the definition of these rules is another complex problem to solve.

While the assignment of new flows to a concrete instance is trivial when using a hash based micro forwarding solution, i.e. the hashing algorithm will take care of that, it is not directly clear what entity would make the decision when using rule based forwarding. Some approaches will forward the packet to the controller/orchestrator that will decide based on load metrics what instance should handle the new flow, which introduces a bottleneck to the system. Other approaches perform the assignment to an instance on the previous NF host (e.g. Stratos). Another possible solution could be

[1] there are also more complex load balancing solutions like Maglev [EYC+16]

to have a default instance that will receive all new flows and either process the new flow or forward it to another instance if itself is busy. The default instance would have to reserve some portion of its capacity for this new flow handling. This concept is used and extended in the switch offloading technique (cf. 5.4).

Another potential solution that completely avoids micro forwarding and solves the new flow handling is to split the flow tags, e.g. instead of a single internal source there are 2 source tags based on some distinguishable property and use the different resulting tags to determine to which concrete instance a flow should be forwarded to. If a number of sub tags, for example the maximum number of possible instances, are defined in advance, the static tag definition could be kept, otherwise support for dynamic assignment of new tags (as FlowTags [FCS+14] originally describes) would have to be supported.

Figure 5.3 shows what steering pattern can be used depending on state management and traffic characteristics. If active load balancing is used (see 5.2), a new flow handling pattern is also required.

Another thing that is also important to consider but is easy to forget is that flows are mostly bidirectional, meaning for a flow $A \rightarrow B$ there is probably also a reverse flow $B \rightarrow A$. NFs often require to process packets from both flows (e.g. NAT, Firewall, IDS) and it is sensible to process both flows on the same instance as state is often scoped for this flow pair[2]. The bidirectional nature complicates chain definition and does not fit well with the DAG abstraction used by most approaches. The solution to guarantee instance affinity across bidirectional flows depends on the micro steering pattern used. Hash-based steering can ensure instance affinity with a symmetric hashing algorithm. For rule and tag based steering the reverse rules will typically be installed at the same time the non-reverse rules are installed [3]. E2 [PLH+15] solves this by using two policy chains that use the same underlying NF while in Stratos chains are defined bidirectional. Other approaches do not explicitly mention handling of bidirectional traffic. The chains in the system model presented in Chapter 4 are defined unidirectional and uses the same solution as E2, i.e. using the same meta-instance in both chains. If an instance processes packets from a bidirectional flow it is also advisable to assign two ports to the instance, otherwise the single port could limit the throughput.

5.4 Switch Offloading

The biggest difference between the proposed system design and existing solutions in literature is the inclusion of switches as platform to execute network functions. While the authors of E2 and OpenBox mention the possibility to offload certain aspects, e.g. classification or header rewrites, to hardware switches, in our proposed design network functions can run in the control plane of the switch and program the data plane flow tables to perform the actual functionality of the NF at line rate, if the network function fits the flow-action abstraction of hardware switches. However, as shown in [KDBR17] and the evaluation of our prototype implementation, the communication

[2] In the literature flow scope is used for both, a single flow or a flow/reverse flow pair

[3] for the rule-based case the macro to micro steering hand off might not be on the same switch for both flow directions, complicating the forwarding

path between switch data and control plane has rather limited performance, which renders switch instances a bottleneck for handling new flows that require handling in the control plane. There is however a practical solution to this problem as well as load balancing, assuming that a switch instance and a server based instance run on directly connected nodes. The switch can let a server implementation perform the processing of new flows and the server instance can decide to assign the flow and install the state to itself or to the switch (or both for fault tolerance). The same solution could be used for load balancing, i.e. the server could decide to offload processing of heavy flows to the switch. The forwarding to an adjacent server instance does not add a noticeable latency or throughput penalty, especially when compared to processing new flows on the orchestrator. The fact that the traffic would pass through the switch anyway allows to migrate a flow or perform a failover from the server to the switch instance without complex forwarding changes to multiple switches, allowing to delegate the decision making to the instances.

6 Pattern Catalog

This Chapter presents the catalog of patterns that were compiled from literature as well as our own considerations. The patterns are based on the system model and design introduced in the previous chapters. Table 6.1 gives an overview of the presented patterns.

Pattern	Aspect	Page
Formulate Placement as Optimization Problem	Placement	46
Unidirectional Chain Definition	Chaining	47
Separation of Macro and Micro Forwarding	Chaining	48
Tag-based Chain Definition and Macro Forwarding	Chaining	49
Simple Migration	State migration	50
Coordinated Migration	State migration	51
Write All / Read One	Fault tolerance	52
Active/Active	Fault tolerance	53
Buffered IO and Snapshots	Fault tolerance	54
Rollback Recovery	Fault tolerance	55
Shared Datastore	Fault tolerance	56
Simple Share	State sharing	57
Serialized Share	State sharing	58
Active Rule-based Steering	Micro forwarding	59
Passive Rule-based Steering	Micro forwarding	60
Hash-based Steering	Micro forwarding	61
Tag-based Steering	Micro forwarding	62
Fallthrough Handling	New flow handling	63
Centralized Handling	New flow handling	64
Local Event Handling	Switch offloading	65
Selective Offloading	Switch offloading	66
Use VNF Frameworks	Implementation	67
Implementation Independent State Representation	Implementation	68
Containerized Deployment	Implementation	69
Consolidation	Implementation	70

Table 6.1: Overview of the pattern catalog

6.1 Topological

6.1.1 Placement

6.1.1.1 Formulate Placement as Optimization Problem

Most discussed approaches (among others Stratos [GGA+12] and Slick [ABFL15]) use linear programming to model and solve the placement of NF instances. This allows to express complex requirements and optimization goals, for example to maximize the total system throughput or to minimize latency and hop-counts or to place all backup instances on the same hosts so they can be powered down in low load situations.

Requirements

- Definitions of Topology, Chains, NFs and their performance and resource requirements/properties

Benefits

- Once a formalized model exists it can be reused for different networks

Drawbacks

- Complex problem, hard to verify

Related patterns Unidirectional Chain Definition

6.1.2 Chaining

6.1.2.1 Unidirectional Chain Definition

The unidirectional definition of chains reduces the complexity and mental overhead for administrators. The same meta-instance of the same NF can be used in different chains, allowing to express inter-chain (and bidirectional) dependencies.

Requirements

- The instance affinity property often also applies to bidirectional flow pairs, requiring that the used steering pattern ensures that the property is adhered

Benefits

- Reduced chain policy complexity compared to bidirectional model

Related patterns Load-balancing and Scaling

6.1.2.2 Separation of Macro and Micro Forwarding

The separation of forwarding between chain elements and the forwarding to a concrete instance simplifies the overall forwarding complexity by splitting it in two simpler sub problems at the expense of latency and efficiency.

Requirements

- Macro and micro forwarding patterns
- Placement algorithm should be aware of the separation
- Fixed hand-off locations between macro and micro forwarding

Benefits

- Reduced complexity
- Separation of concerns
- Limits the involved entities to solve instance affinity

Drawbacks

- Latency overhead
- Possible introduction of bottlenecks, e.g. the traffic for all instances must pass through the same switch port

Related patterns Tag-based Chain Definition and Macro Forwarding, Load Balancing and Scaling

6.1.2.3 Tag-based Chain Definition and Macro Forwarding

Using coarse-grained tags attached to packets to mark the high level packet source and destination, processing state and metadata, allows to define abstract, expressive and easy to maintain chain policies (cf. FlowTags [FCS+14], E2 [PLH+15] and Chapter 5). Tags allow to define conditional processing steps, provide an elegant solution to deal with mangling NFs and reduce the number of forwarding rules required but requires modifications to NFs.

Requirements

- Tag support in NFs and gateway switches
- Orchestrator must compute valid tag mapping

Benefits

- No flow correlation technique (e.g. SIMPLE [QTC+13]) is required to deal with mangling NFs
- Allows to define high-level chain policies that can easily be reused for different concrete networks
- Allows to use conditional forwarding based on NF output with the same abstraction
- Reduces number of required rules in the network

Drawbacks

- NFs must be modified to support processing state and metadata tagging
- Used header field might limit tag usage, e.g. VLAN only supports 4096 different values

Related patterns Separation of Macro and Micro Forwarding, Tag-based Steering

6.2 State

6.2.1 State migration

6.2.1.1 Simple Migration

A simple migration moves state associated to some scope from one instance to another without accounting for additional NF network view semantics. OpenNF [GVP+14] and Split/Merge [RWJW13] describe such a move operation but differ slightly in their implementation. On an abstract level the old instance must take a snapshot of the state that has to be moved and stop to process further packets at the same instance. The state is transferred to the new instance and once the new instance is ready, the forwarding rules are updated to point to the new instance.

Requirements

- Rule-based Steering
- Implementation Independent State Representation

Benefits

- Enables active, fine-grained load-balancing
- Simple sequence of operations, easy to implement

Drawbacks

- Will drop packets during the move and in consequence cause retransmissions[1]

Related patterns Load Balancing and Scaling, Implementation Independent State Representation, Coordinated Migration

[1] Split/Merge buffers packets at the controller during the move and sends them to the new instance after the move, which can still cause retransmits it the move takes too long.

6.2.1.2 Coordinated Migration

A coordinated move can be used to migrate state between instances while ensuring drop-free and order-preserving network view semantics for the network function instances, i.e. the network function will see the same packets in the same order if there would be no state migration as described in OpenNF [GVP+14]. To enable a coordinated migration, the orchestrator (or a dedicated node in the system) must be able to buffer packets as well as the new instance during the move. The coordinated migration is described in more detail in Chapter 3.

Requirements

- Rule-based Steering
- Implementation Independent State Representation
- Orchestrator must be connected to the production network
- Buffering at the orchestrator and new instance

Benefits

- The move is transparent from the perspective of the NF, which, in the case of an IDS, possibly avoids false negative alerts.

Drawbacks

- If the move takes too long and therefore the buffering, endpoints may retransmit packets
- For a limited duration traffic has to be cloned and forwarded to the old instance and orchestrator
- Increased latency due to the buffering

Related patterns Load Balancing and Scaling, Implementation Independent State Representation, Simple Migration

6.2.2 Fault tolerance and Shared State

6.2.2.1 Write All / Read One

Write All / Read One (ROWA) is a well known quorum based replication protocol in distributed systems literature (cf. [TS06, p. 312]). ROWA is the once extreme end of possible quorum based solutions that trades the lowest possible per-read overhead with the highest per-write cost. ROWA allows a read operation to only consider the local state but requires writes to be performed on all replicas. This trade off however fits well for state that is only written once.

Requirements

- Write Once / Read Per-Packet state access pattern
- Implementation Independent State Representation
- It must be ensured that the output commit property is enforced

Benefits

- Allows to failover and migrate between instances
- Simple to implement
- Does not require buffering

Drawbacks

- Requires that all instances are available
- Wastes limited flow table capacity on NFO switch instances

Related patterns Implementation Independent State Representation, Selective Offloading

6.2.2.2 Active/Active

Two identical instances process the same slice of traffic, while the output of one of the instances will be dropped until the other instance crashes. Assuming the instances behave like a FSM and end up in the same state given the same input, they should remain identical copies. After a crash, an additional update protocol would be required and in practice determinism seems like an unrealistic assumption, as most NFs are multithreaded and/or use timeouts [SGB+15], which are typical sources of non-determinism.

Requirements

- Traffic duplication
- Deterministic NF

Benefits

- Supports any state access pattern and scopes
- No communication between the two instances required

Drawbacks

- Can lead to inconsistent state if processing is non-deterministic or packets are dropped on one of the links after duplication
- High resource (duplicated traffic, instances) usage

Related patterns Rollback Recovery

6.2.2.3 Buffered IO and Snapshots

Input and output buffers are used to delay the release of processed packets and processing of new packets until the associated state has been transferred to another instance. This pattern is described in Pico replication [RWJ13].

Requirements

- Rule-based Steering
- Implementation Independent State Representation
- Input and output buffering at every instance
- It must be ensured that the output commit property is enforced

Benefits

- Supports sharding of state
- Can deal with write-per-packet access pattern
- Can be used to share state with either eventual (merge-based) or strong consistency (with correspondent throughput penalty)
- Snapshot frequency can be configured on a per-flow (scope) level

Drawbacks

- Increased latency
- Generates bursty traffic

Related patterns Rollback recovery, simple share, serialized share

6.2.2.4 Rollback Recovery

Rollback recovery, as described in FTMB [SGB+15], provides fault tolerance by recording incoming packets and the ordering of state accesses on a remote node, starting from a periodically updated state snapshot. In a failover event the backup instance will load the latest snapshot and process the recorded input packets in the exact order that was recorded.

Requirements

- It must be ensured that the output commit property is enforced
- Specialized Topology and Input/Output logger nodes
- Buffering of input packets for the duration since the last snapshot
- Buffering of output packets until their state accesses have been saved

Benefits

- Reduced latency overhead compared to the buffered IO model
- Can deal with non-deterministic NF behavior
- Provides fault-tolerance

Drawbacks

- Requires numerous complex modifications to NFs
- Does not provide load-balancing as only a single instance is active

Related patterns Buffered IO and Snapshots, Active/Active

6.2.2.5 Shared Datastore

The shared datastore pattern equates to the StatelessNF [KCH+15] system. Instances are effectively stateless and will interact with a replicated, remote key-value datastore that holds state. Datastore requests are batched to amortize the remote operation overhead which slightly increases latency.

Requirements

- Dedicated datastore servers
- RDMA setup (Infiniband) and specialized datastore
- Implementation Independent State Representation
- It must be ensured that the output commit property is enforced

Benefits

- Allows to use hash based or round-robin steering
- State and processing are separated, which allows to scale instances without moving state or accounting for instance affinity and enables fault tolerance

Drawbacks

- Unpredictable latency depending on state hit/miss on local datastore instance, possibility for DOS attack
- Higher resource requirements
- Timeouts do not fit the key-value datastore abstraction[2]

Related patterns Buffered IO and Snapshots, Simple Share, Serialized Share, Hash-based steering

[2] In [KCH+15] this is solved by adding timeout callback support to RAMCloud

6.2.2.6 Simple Share

The simple share pattern provides a way to share state between multiple instances with eventual consistency semantics. Both, OpenNF [GVP+14] and Pico replication [RWJ13] describe a way to implement a simple share operation. When a packet modifies shared state, the state will be copied to another instance (or instances) where a NF provided function will merge the local and remote state according to the NF semantics.

Requirements

- Implementation Independent State Representation
- It must be ensured that the output commit property is enforced
- NF specific state merge function

Benefits

- Low overhead compared to serialized share

Drawbacks

- NF must be able to work with eventual consistent state

Related patterns Serialized Share

6.2.2.7 Serialized Share

The serialized share pattern describes a solution to apply updates to shared state in the same order on all instance holding that state, in the context of some state scope. OpenNF [GVP+14] proposes a solution where packets, that would modify shared state, are send from the instance to the orchestrator which will define the global order, send the packet back to the origin instance where it will be processed and the resulting state update will be copied to other instances, before the next packet in the handled scope is handled in the same way.

Requirements

- Implementation Independent State Representation
- It must be ensured that the output commit property is enforced
- Buffering at the orchestrator

Benefits

- Serialized ordering of updates

Drawbacks

- Increased latency due to the buffering
- Reduces throughput as packets and state updates are handled in lock step

Related patterns Simple Share

6.3 Dynamic

6.3.1 Load Balancing and Scaling

6.3.1.1 Active Rule-based Steering

Active rule-based steering will perform fine-grained load balancing on the flow level, i.e. migrate a flow from one instance to another, as described in Split/Merge [RWJW13] and OpenNF [GVP+14]. State must be moved with the flow.

Requirements

- Solution for state migration

Benefits

- Allows to drain instances from maintenance tasks

Drawbacks

- Does not scale well as many fine-grained rules have to be maintained due to the limited flow table capacity

Related patterns State-migration, Passive Rule-based Steering

6.3.1.2 Passive Rule-based Steering

Passive rule-based steering will only perform coarse-grained load-balancing in the sense that exiting flow assignments are not migrated and therefore does not require a state migration pattern. New flows will be assigned to a new instance by a new flow handling pattern (cf. 6.3.2). To save flow table space, coarse-grained forwarding rules are usually used for micro-steering and may map new flows to an old instance which usally is solved by using exception rules. If this is the case, instances can be daisy-chained as in E2's migration avoidance algorithm [PLH+15] or the fallthrought handling pattern. The old instance will then forward packets of flows it has no associated state to a new instance, which avoids a possibly large number of (per-flow) exception rules.

Requirements

- Support for daisy-chaining in the network function

Benefits

- Requires no state migration pattern
- Daisy-chaining (flow migration avoidance) can reduce the flow-table usage

Drawbacks

- long lived flows can cause inefficient resource usage
- Inefficiencies (increased latency, resource usage) by forwarding traffic from instance to instance (if used)

Related patterns Fallthrough Handling, Active Rule-based Steering

6.3.1.3 Hash-based Steering

Hashing provides a solution to the micro-forwarding problem introduced by load-balancing and scaling. While resilient hashing algorithms remain mostly stable, they still reassign a number of hosts during a scale-up event, limiting the applicability to stateless or fully replicated solutions. Bidirectional flows can be handled with symmetric hashing solutions. Most of this also applies to simple round-robin based schemes.

Requirements

- Network function must either be stateless or the state must be available on all instances
- Support on the switch hardware[3]
- Hashed header fields must be a superset of the state scope

Benefits

- Scales well as no per-flow state on the switch is required
- No special new-flow handling required
- The hash buckets can often be given a weight to account for differences in instance performance

Drawbacks

- No influence on flow assignment outside of weighting

Related patterns Shared Datastore

[3] Optional in the OpenFlow switch specification

6.3.1.4 Tag-based Steering

The source or destination component of a flow tag is split in a number of sub-tags which are used to map packets/flows to NF instances.

Requirements

- Tag values must be a superset of the state scope
- Good sub-tag definition

Benefits

- Reduces overall system complexity
- Does not require micro-steering

Drawbacks

- Does not support dynamic scaling
- If the property used to define the sub-tags does not uniformly distribute the traffic, instances can be overloaded

Related patterns Fallthrough Handling, Central Handling, Tag-based chain definition

6.3.2 New Flow Handling

6.3.2.1 Fallthrough Handling

Fallthrough handling is inspired by the migration avoidance algorithm presented in E2 [PLH+15] (cf. 6.3.1.2). Packets of new flows are forwarded to a designated instance of each NF meta-instance in the current chain, which will either process the packet and trigger the installation of forwarding rules that assigning the flow to itself or pass it on to another instance of the same NF meta-instance depending on it's own utilization. In the case of a NFO switch based instance, fallthrough handling does not reduce the performance, unlike the server to server case.

Requirements

- Rule-based steering pattern
- In-NF chaining support, i.e. instances must know each other and how to pass packets to a certain instance

Benefits

- Simple solution, requires few rules

Drawbacks

- Increased latency for new flows
- Consumes processing capacity on server instances
- Possible single point of failure and denial-of-service target

Related patterns On Switch Handling, Centralized Handling, Local Event Handling

6.3.2.2 Centralized Handling

Packets of a new flow are sent to a system global instance (e.g. the orchestrator) that will determine the instances of every NF in the chain that the flow will be assigned to. The centralized planning can optimize the placement based on inter-instance traffic and load statistics. Among others, Stratos [GGA+12], Split/Merge [RWJW13] and SIMPLE [QTC+13] use this method.

Requirements

- Rule-based steering
- Orchestration connected to the production network

Benefits

- Can optimize inter-instance traffic flows

Drawbacks

- Possible single point of failure and denial-of-service target

Related patterns On Switch Handling, Fallthrough Handling, Selective Offloading, Topology Aware Placement

6.3.3 Switch Offloading

6.3.3.1 Local Event Handling

When using a write all / read any state replication protocol, with a switch and server implementation, the switch implementation can install the processing rules with a lower priority than the forwarding rule to an instance. In the case of a crashed server instance, the failover operation is as simple as removing the forwarding rule to the failed instance. Compared to a failover reaction where potentially multiple switches have to install new forwarding rules to another instance that hold the state of the failed instance, that also have to be computed first, the reaction time can be reduced.

Due to the limited capacity of switch forwarding rules, it might make sense to combine this pattern with the next pattern, selective offloading, and only perform the on-switch handling for a subset of important flows and still perform the normal failover handling.

More involved fault tolerance patterns should be used if state size, access pattern or share pattern no not fit the listed requirements.

Requirements

- NFO capable switch and network function (fits the match-action model)
- Placement of instances on directly connected NFO switch and NF server
- Write-Once / Read-per-packet state access pattern
- Write All / Read Any state share pattern

Benefits

- Reduced reaction time compared to global event handling
- After the server instance recovers or a new instance was started, the state can be recovered from the switch instance

Drawbacks

- Limited flow table capacity, can result in partial handling of flows after failover event
- Only works for a server crash, a switch crash requires global forwarding changes

Related patterns Fallthrough Handling, Selective offloading, Write All, Ready Any

6.3.3.2 Selective Offloading

While most flows in a data center network are relatively short lived (seconds) and transfer only a small number of bytes [BAM10], some flows live long and have a high volume (e.g. backups). By moving such flows to an instance running on a NFO switch, load on instances running on a NF server can be reduced. A switch implementation can process a large flow with no negative impact on the other flows, while a server based implementation typically has to spend more time and processing power, resulting in reduced service quality for other flows. However switch instances are limited in their state (flow tables) capacity, limiting the number of flows that can be processed. When the network function state is only written once, offloading is as simple as installing a higher-priority rule on the switch. Theoretically no packet drops should occur as packets are either send to the server instance that will continue to hold the state (and still *owns* the flow), or the new switch rule, which also applies for the reverse action. Note that timeout handling, if done by the NF, also has to be transferred to the switch.

Requirements

- NFO capable switch and network function
- Placement of instances on directly connected NFO switch and NF server
- Write-Once / Read-per-packet state access pattern

Benefits

- Reduced load on server based instances caused by heavy flows
- Line rate processing
- Only local, isolated change to forwarding
- Can also be used for flows with low-latency requirements

Drawbacks

- Limited flow table capacity
- Packet reordering during move

Related patterns Fallthrough Handling, Local Event Handling

6.4 Implementation

6.4.1 Technical

6.4.1.1 Use VNF Frameworks

There are several frameworks (cf. section 3.4) that provide useful APIs and abstractions for developing VNFs. These frameworks usually are based on a packet processing framework like DPDK but take care off common tasks like initialization or provide more useful abstractions, for example mOS [JMK+17] allows to work with reconstructed TCP bytestreams and Netbricks [PHJ+16] offers a high level iterator based API that transparently takes care of batching and packet IO. By using a framework that fits the abstraction and operation of a NF, the development and maintenance cost can be reduced since the actual functionality can be in the focus. If multiple (custom) network functions are to be developed or integrated with an orchestration solution, an existing framework can easily be extended to support the orchestration (metrics, control events, fallthrough pattern) or other environment specific tooling, e.g. configuration management or event tracing.

Requirements

- Deployment environment that is compatible with the framework (e.g. supports DPDK)
- Framework should be compatible with the state representation (cf. 6.4.1.2)

Benefits

- High level abstractions hide complexity of underlying packet processing
- Reduced development and maintenance cost

Drawbacks

- Have to check that framework fully supports the use case, which might not be easily determinable. Otherwise modifying the framework or moving the implementation to another framework can be complex and expensive[4].

Related patterns Implementation Independent State Representation

[4] For the prototype implementation Netbricks was extended with a custom iterator type which required some time to understand the inner workings of the framework

6.4.1.2 Implementation Independent State Representation

The strict performance requirements can cause in-memory state representations to become highly optimized. For example logically dependent state might be split up to fit into a single cache line or to be processed by different cores. State migration, sharing and fault tolerance patterns require to move the state between instances, which is not a problem as long as there is only a single implementation. When running different versions or implementations (e.g. for an ONFSwitch), the specific data representation might change or differ. Therefore, it is advisable to use a more general representation for state when moving or copying state between instances. Many different data exchange formats exist, for example Protocol Buffers, FlatBuffers or Thrift use a binary encoding, but there are also text based formats like JSON or XML and many language specific solutions (which should only be used if it is cleat that only a single implementation language is used).

Benefits

- State can be moved between different implementations
- Sharing state with other systems or the orchestration is straightforward
- Debugging is simplified as most data formats have a simple human readable format
- Persistent storage in disk is trivial to add if needed

Drawbacks

- Slight performance drawback for (de)serialization and conversion compared to custom binary protocols

Related patterns Use VNF Frameworks, State Migration, High-Availability, Shared State

6.4.1.3 Containerized Deployment

Many of the presented orchestration and framework approaches use containers or VMs as deployment format for NFs and as a way to provide isolation between different instances that run on the same NF server. For the NF use-case containers appear to better fit. Running a VM only to execute a single binary compared to the more lightweight kernel level isolation of containers seems excessive. NIC (Ports) can be passed to a container so that a NF in a container can still use DPDK for fast packet processing.

Benefits

- Allows to use existing compute orchestration frameworks
- Provides isolation between instances running on the same host without the overhead of a VM

Drawbacks

- Possible incompatible with switch control plane OS
- Configuration overhead when using DPDK

6.4.2 Conceptual

6.4.2.1 Consolidation

For NFs that are always deployed together, for example an external firewall and NAT, or NFs operate on the same higher abstraction like a reconstructed TCP byte-stream, it can make sense to consider consolidating multiple NFs into a single entity. Consolidated NFs reduce orchestration overhead as there are fewer entities to consider, as well as latency from redundant processing steps can be eliminated. OpenBox [BHH16] and CoMb [SER+12] describe how NFs can be consolidated in a way that preserves separation between the different processing steps.

Benefits

- Reduced orchestration complexity, i.e. fewer entities to consider
- Improved efficiency by elimination redundant parsing and serialization steps

Drawbacks

- Possible large and complex composite NFs
- If modularization is not strictly enforced, the implementation can become hard to maintain and separating the different NFs later on can be difficult
- Debugging the behavior of a consolidated NF (e.g. performance) might be more involved
- Scaling, HA and state of a consolidated NF are also combined, possibly multiplying the complexity

7 Implementation

This chapter presents the implementation of a distributed NAT system, based on the design and patterns from the previous chapters. NAT was chosen as it is a widely used NF, that supports switch offloading. We will first discuss the scope of the prototype, then the high-level architecture and synchronization mechanism are presented, as well as the components and workflow of the switch and server NAT systems.

The prototype implements the following patterns:

- Write All / Read One state synchronization
- Fallthrought handling for new flows
- Local failover event handling
- Selective Offloading
- Implementation using a VNF Framework
- Implementation Independent State Representation

Rust [rust18] was chosen as implementation language to leverage NetBricks [PHJ+16], a framework for writing network functions, for the server NAT implementation. Rust is a systems programming language that originates from Mozilla. It offers the same low-level control of the classic systems programming languages like C and C++, but ensures memory and thread safety. These guarantees are enforced at compile time, allowing Rust to achieve performance similar to C and C++. At the same time Rust offers many features and language constructs that were previously only available in higher level languages like flexible iterators, a powerful type system, integrated unit tests, sane dependency management and facilities for easy (de)serialization of data. Rust also offers an efficient foreign function interface (FFI), that allows the use of libraries written in C without overhead from within Rust code. The switch implementation for example uses the LOCI OpenFlow library [loci18] to create and parse OpenFlow messages. The server and switch NAT are implemented in about 4500 lines of Rust code. NetBricks, for comparison, in about 6800 lines[1].

7.1 Scope

As already mentioned, due to the prototypical nature and timing constrains, the overall proposed design was only implemented partially:

[1] Both measured with cloc, ignoring generated code

Orchestrator The orchestrator was not implemented as it does not concern the core functionality of the prototype implementation. For the evaluation the configuration was loaded from local configuration files [2].

State Merge When an instance reconnects after a failover event, only one of the peers is allowed to hold NAT mapping state. For a server NAT crash, this will typically be the case, however in other scenarios this might still happen. If processes detect state on both processes, they will terminate.

DNAT While the data model and prototype architecture support DNAT, the server and switch NAT do not implement it. Adding it should be a straightforward task.

Same Implementation instances The implementation only supports a pair of switch and server implementation. While same implementation setups would be able to connect and share state, the orchestrator would be required to fully utilize these scenarios (e.g. network configuration).

Single port mode The switch implementation supports a peer that is only connected by a single link and the IP DSCP field to tag the traffic as internal or external origination, instead of separate links for internal and external traffic. The server implementation does not implement this mode.

7.2 Architecture

The overall system is depicted in Figure 7.1. The switch NAT application, running in the control plane directly on the switch, configures the data plane via the OpenFlow protocol. The Pica8 switch that was used during implementation and evaluation uses Open vSwitch (OVS)[ovs18] to implement the OpenFlow protocol. PicOS then maps the OVS configuration to the actual switch data plane. The server implementation runs on a standard Linux x64 server. As already mentioned, NetBricks was used to implement the packet processing. Netbricks offers a convenient, higher level framework based on DPDK[dpdk18], a high performance user space networking framework. The server and switch data plane are connected by two links, mainly to support full line rate in both NAT directions, i.e. internal to external and vice versa. Both applications also use an out of band control channel for state synchronization.

Both implementations are multithreaded and use a message (event) passing for communication between threads.

[2] This can impact the correctness after a failed process reconnects after a failover: The process would load his previous NAT pool from the configuration file, but might receive mappings from that pool during the initial state transfer and would not update it's pool state

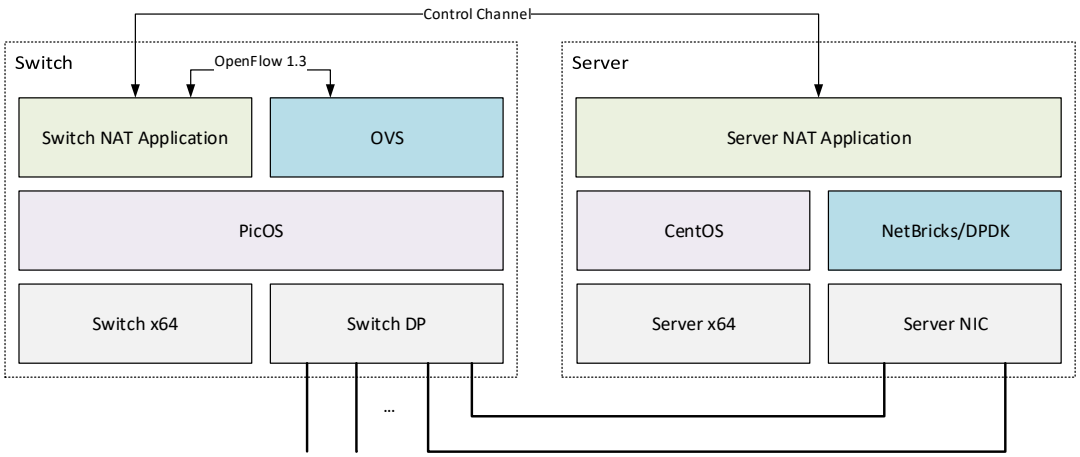


Figure 7.1: High level overview of the implemented system

Event	Fields	Description
Ready		The initial configuration of the switch dataplane has completed, i.e. the default set of flows is installed
TransactionComplete	`txid: u64`	A requested change to the dataplane has completed, contains the TXID of the request
Error		The switch responded with an OpenFlow error message. This will propagate to the peer and cause the process to terminate
FlowTimeout	`cookie: u64` `entries: Vec<NatEntryPair>`	A flow rule timed out
PacketIn	`packet: Packet`	A packet was received from the dataplane, i.e. a new flow for that no NAT entry exists

Table 7.1: Switch connection module events

7.2.1 Switch NAT

The switch implementation is structured in three modules, as shown in Figure 7.2. The switch and peer connection modules handle the communication with the switch and peer process respectively. The main event loop in the control plane module processes the events it receives from the two other modules. The events (messages) that the peer connection module sends to the core are listed in Table 7.3. The switch connection module can send the events listed in Table 7.1.

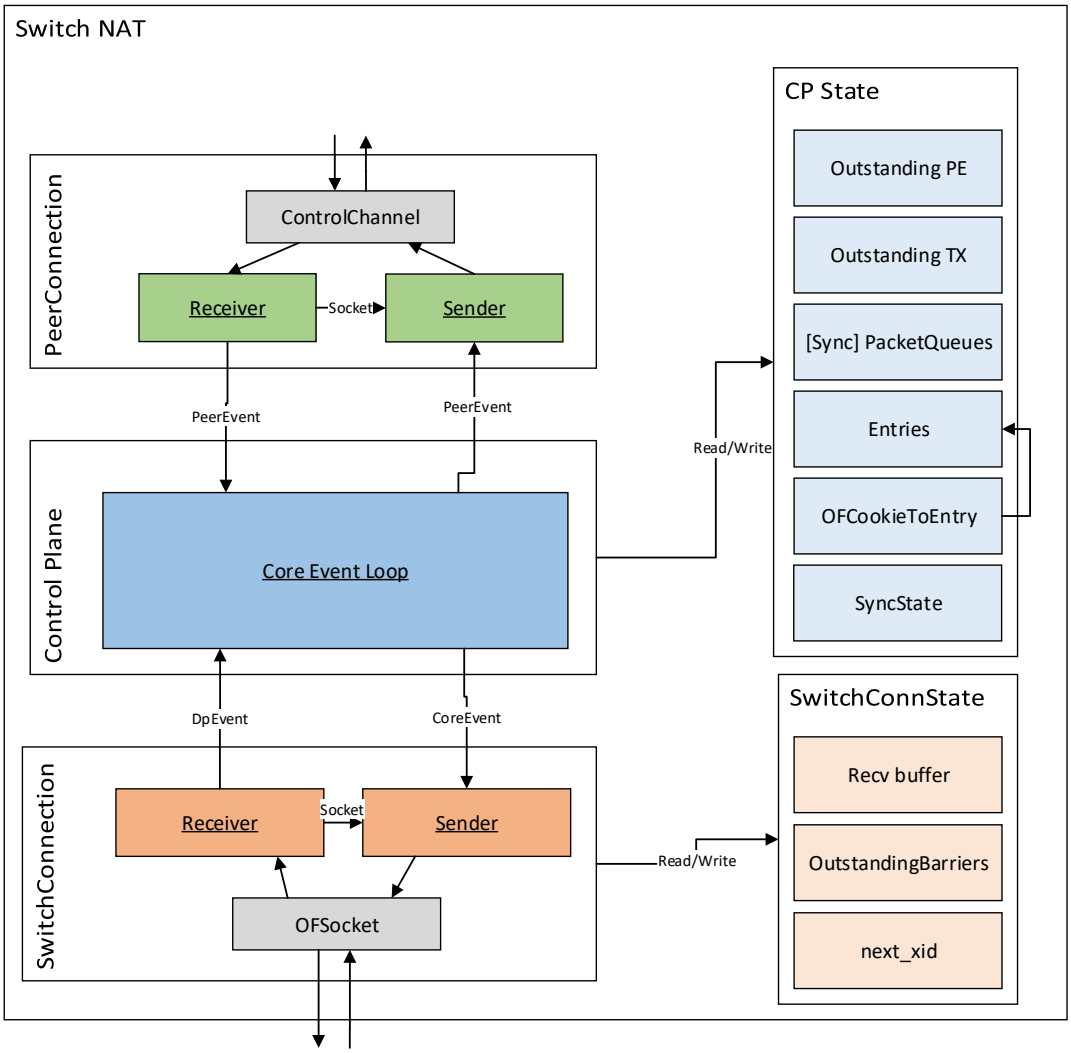


Figure 7.2: Switch NAT architecture. Underlined components mark a running thread

The core uses the peer event messages listed in Table 7.3 and the events listed in Table 7.3 to communicate with the switch connection layer.

All but the PacketOut event contain a TXID that the core will use to map a response from the switch connection module to the source of the event, for example a peer transaction, to know what to do when receiving a `TransactionComplete` event.

The switch connection module will translate the high level event representations to OpenFlow messages and replace abstract port definitions with the actual switch ports as defined in the configuration. For flow installations or removals a `BARRIER_REQ` is issued and a `TransactionComplete` event is only sent after receiving the matching `BARRIER_REP`. The module further takes care of using the correct timeout and priority for all installed flows according to the configured partition mode

Event	Fields	Description

InstallFlowVecs	`txid: u64`
	`internal: Vec<NatEntry>`
	`external: Vec<NatEntry>`
		Install a batch of NAT entry pairs

InstallFlowPair	`txid: u64`
	`internal: NatEntry`
	`external: NatEntry`
		Install a single NAT entry pair

DeleteFlow	`txid: u64` `cookie: u64`	Delete a single flow

DeleteFlowPair	`txid: u64`
	`cookie_internal: u64`
	`cookie_external`
		Delete a NAT entry pair

PacketOut	`packet: Packet` `action: NatAction`	Inject a Packet into the dataplane and apply the given action
InstallStaticPeerRules	`txid: u64`	Install the forwarding rules to the peer
DeleteStaticPeerRules	`txid: u64`	Delete the forwarding rules to the peer

Table 7.2: Core to switch connection module events

(all local , all peer, new peer, L4 protocol) and timeout rules (detault mode is that only the source process of a NAT entry will timeout it, which for example in the new peer partitioning mode would timeout entries early).

7.2.2 Server NAT

The server implementation is, like the switch implementation, structured into three modules (see Figure 7.3, threads are underlined, lock protected data structures are italic). The peer connection module is identical to the switch implementation and the control plane core event loop is mostly identical to the switch side, but has an additional timer that is used to check for flow timeouts. The third module contains the actual forwarding pipelines. The NAT mappings are kept in a compact representation in two different dictionaries, one for each pipeline (internal and external). Peer updates and timeouts are directly applied into these dictionaries from the core event loop. The internal pipeline will send new flow events to the core loop and, in synchronized mode, also packets of new flows before the mapping action is set to forward. The internal pipeline will also update the external dictionary when it encounters a new flow. Figure 7.4 shows the processing steps for internal packets. Packets are processed in batches of 32 and to improve the performance of the server implementation, a compound operation, that allows to modify and filter packets in a single pipeline step, was added to Netbricks normally filtering (i.e. dropping) and modification would

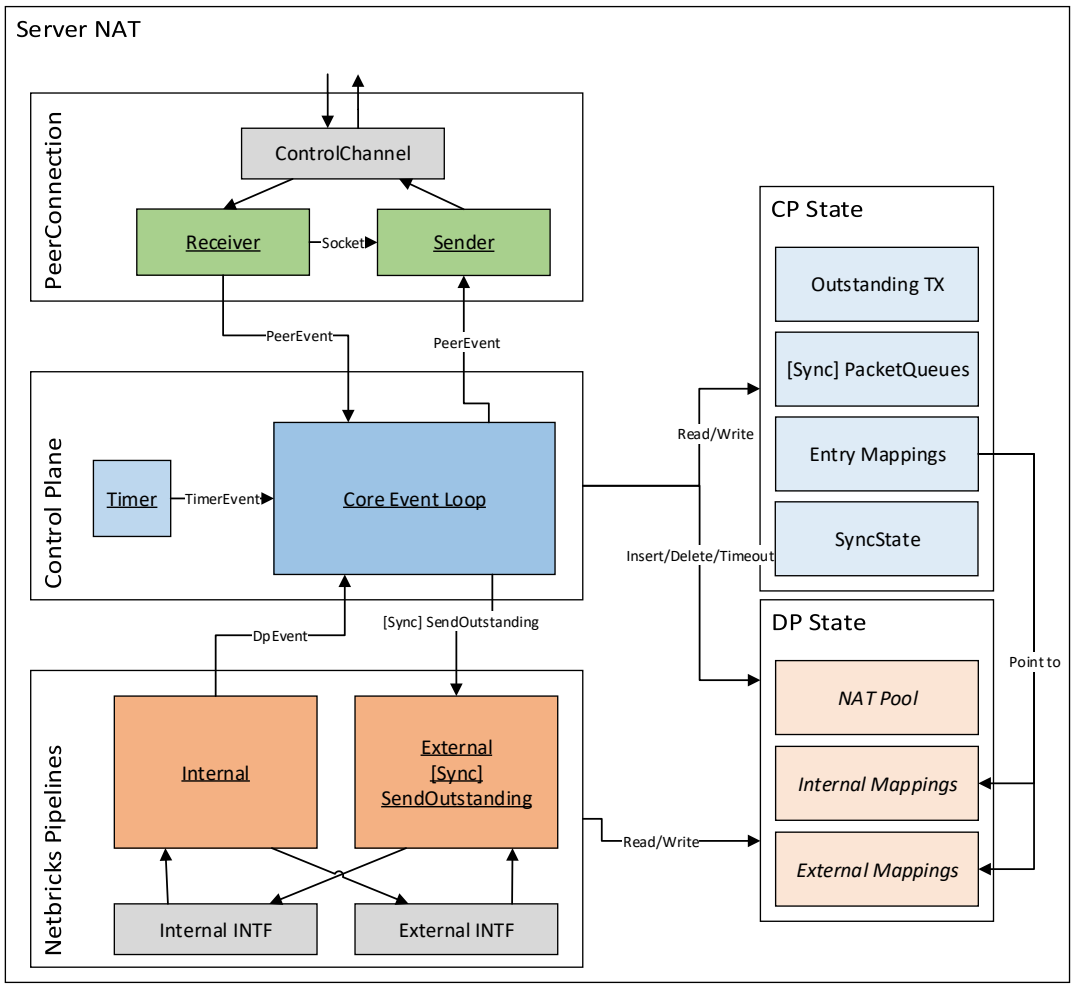
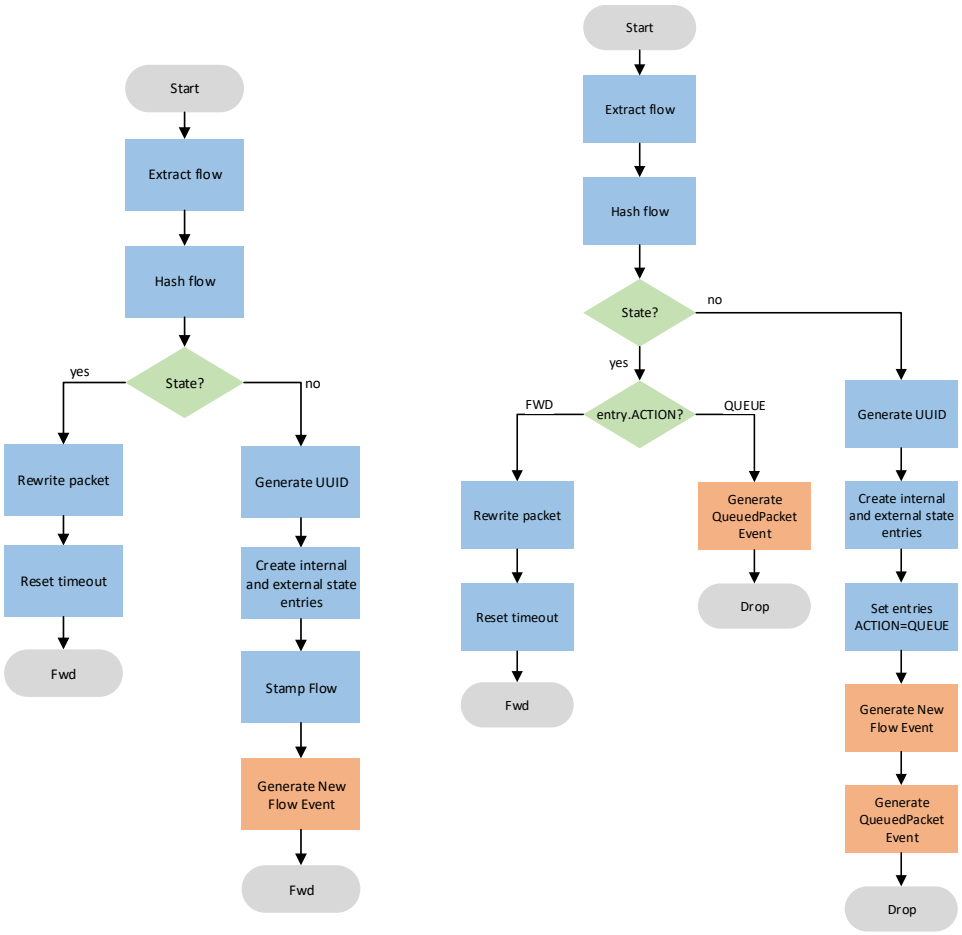


Figure 7.3: Server NAT architecture

have to be performed in seperate pipeline steps. The default hashing algorithm used by Rust was also replaced with FNV hashing [rfnv18], which is faster for smaller key sizes [rhp15], which is the case as the forwarding tables are indexed by flow 5-tuples (13 bytes[3]).

[3]Mac addresses are also included in the low-level flow representation, but they are not included in the hash calculation



(a) Disabled and asynchronous synchronization mode

(b) Synchronous synchronization modes

Figure 7.4: NAT server internal processing pipelines

7.3 Data Model

NAT mappings are stored and shared in a high level representation. Each implementation will translate the entries to a suitable format for their particular dataplane. A NatEntry consists of a match and an action. The match contains the 5-tuple[4] that identifies packets of a specific flow. The action describes how these fields should be rewritten. NAT entries are always stored pairwise, one for each direction and each pair is assigned an UUID. A graphical representation is given in Figure 7.5.

[4]Due to a limitation of the switch hardware, NAT flow rules must also match against a Mac address and VLAN tag. The L2 addresses are therefore included in the NatMatch and Action, which has the advantage that L2 rewrites does not have to be performed externally.

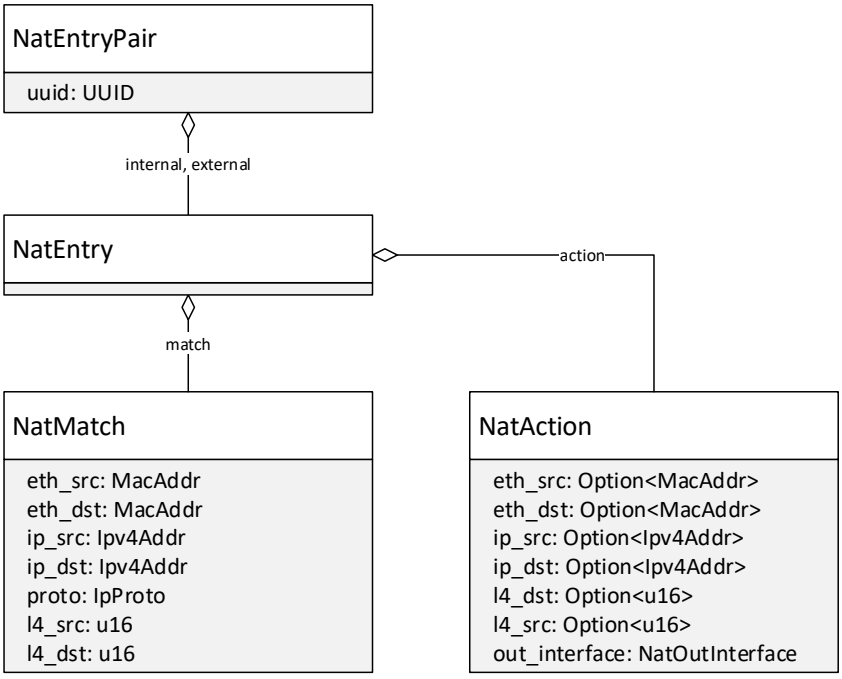


Figure 7.5: High level NAT entry representation

Additionally a pool of free IP/Proto/Port triples is maintained by each process. Because the Orchestrator, that would ensure these pools are disjoint, was not implemented, this property has to be manually ensured. Otherwise duplicate assignments of the same triplet would lead to incorrect behavior.

7.4 State Synchronization

The processes can, depending on the configured synchronization mode, share the NAT entries using a simple synchronization protocol over a TCP connection.

7.4.1 Synchronization Protocol

The protocol message types are listen in Table 7.3. Messages annotated with a * are not serialized but generated by the local connection module. Rust's Serde [serde18] and Bincode [bincode18] crates are used to (de)serialize Rust structs. Bincode guarantees that serialized messages are smaller or equal to their in memory representations.

Each process maintains a state machine, shown in Figure 7.6. After the dataplane layer sends a **Ready** event, processes will wait for a connection from a peer, while handling events locally as in the Off synchronization mode. Once a connection is established, each process will send its current state with a **InitState** message to the peer and wait for the peer's **InitState** message.

Message	Fields	Description
Connect*		Peer has connected
Ready	`txid: u64`	Peer has applied state from a **InitState** message and is ready to process packets
Error		An error occurred, both peers will terminate after sending/receiving an error
InitState	`txid: u64` `entries: Vec<NatEntryPair>`	Initial state send after a connect event
InstallEntries	`txid: u64` `entry: NatEntryPair`	Notification of new flow state
DeleteEntries	`txid: u64` `uuid: Uuid`	Notification of expiration of flow state
InstallComplete	`txid: u64`	Confirmation that a **InstallEntries** message was processed
DeleteComplete	`txid: u64`	Confirmation that a **DeleteEntries** message was processed
Disconnect*		Peer has disconnected

Table 7.3: State Synchronization Messages

Once the received **InitState** message was processed, the process will reply with a **Ready** message and transition into the Operational state. If the dataplane layer sends events during the time from sending the current state to receiving the peer's **Ready** message, these events are buffered and only applied and sent to the peer after the **Ready** message was received. All events not shown lead to an Error state, in which the process will first send an Error message to the peer process and terminate after. In the Operational state, events from the peer and dataplane are processed according to the synchronization mode. If the sync connection is disrupted, the process transitions into the disconnected state and will discard any outstanding operation state. If the connection is restored, the initial two way handshake is repeated.

7.4.2 Synchronization Modes

Three different modes exist for synchronization:

Off State synchronization is disabled, no NAT entries are shared between processes

Asynchronous New entries get shared, the packets of a new flow are immediately sent out after the local entries have been updated

Synchronous New entries are shared and the packets of a new flow are buffered until the peer has acknowledged the new entry

Figure 7.7 shows the difference between asynchronous and synchronous modes.

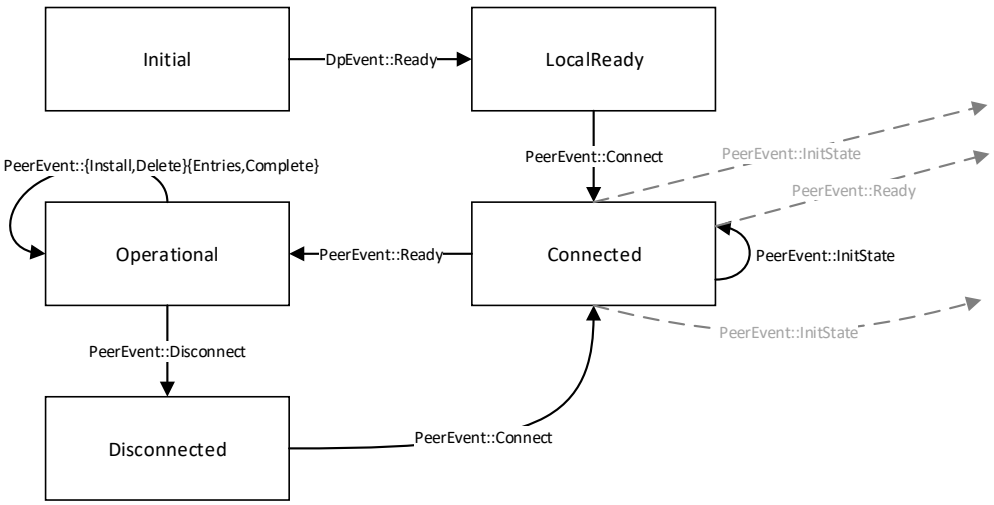


Figure 7.6: State machine of the sync mechanism

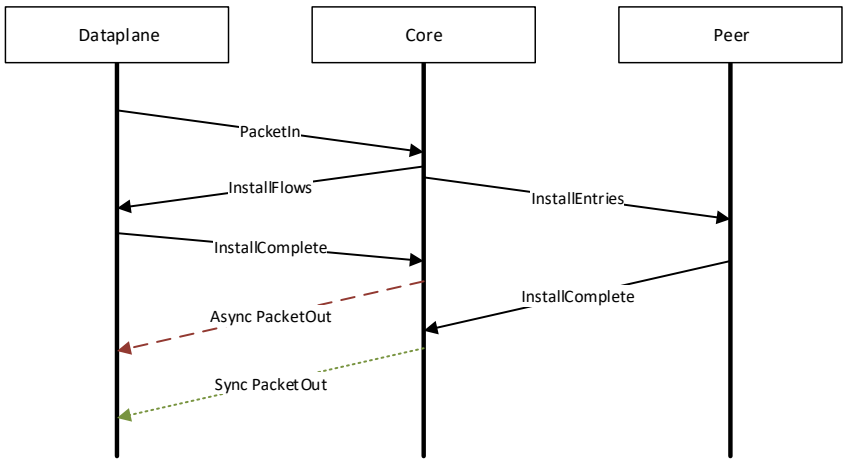


Figure 7.7: Difference between asynchronous and synchronous modes

Note that in the asynchronous case it would not be possible to send the PacketOut without waiting for the InstallFlow to complete, violating the output commit property. This could lead to incorrect behavior, where the external to internal flow is not yet installed and a reply is received for the generated PacketOut[5], or to missing state after a failover event.

Additional synchronization modes could be added with moderate effort, for example a two phase commit.

[5] This mostly applies to the switch implementation

8 Evaluation

The distributed NAT implementation presented in the previous chapter was evaluated for the following aspects:

Throughput How many packets per second can the switch and server implementations handle?

Latency What latency overhead do the implementations introduce?

State Synchronization How do the different synchronization strategies influence latency and throughput?

Fail over How does the implementation behave during a fail over event from the server NAT to the switch NAT?

Micro Benchmarks How long do different internal operations take?

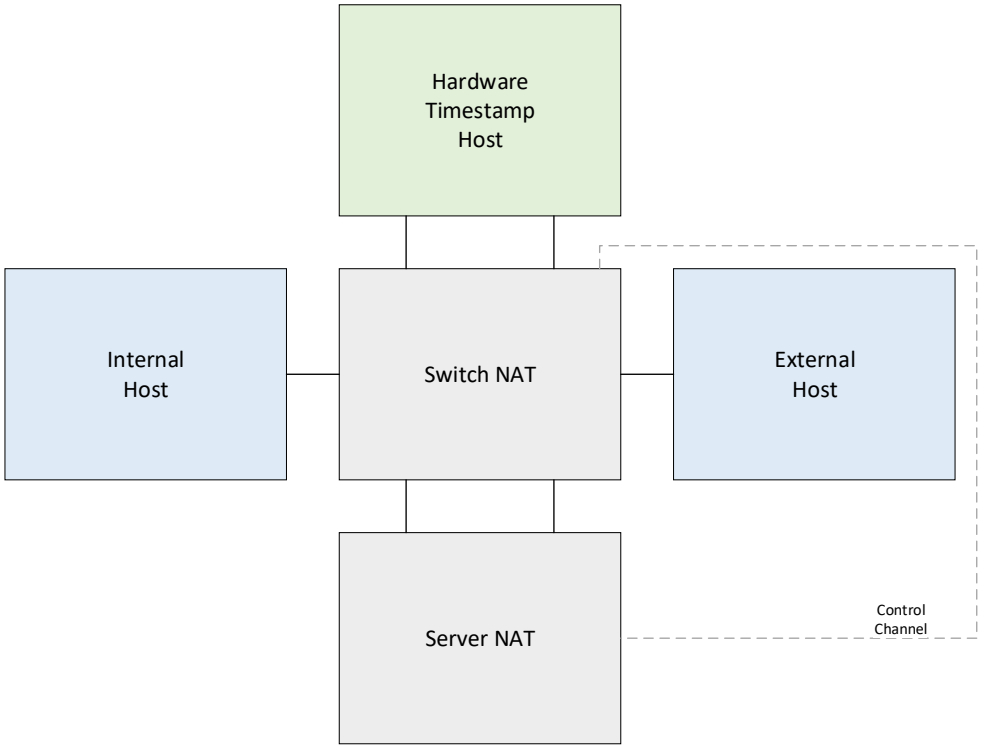
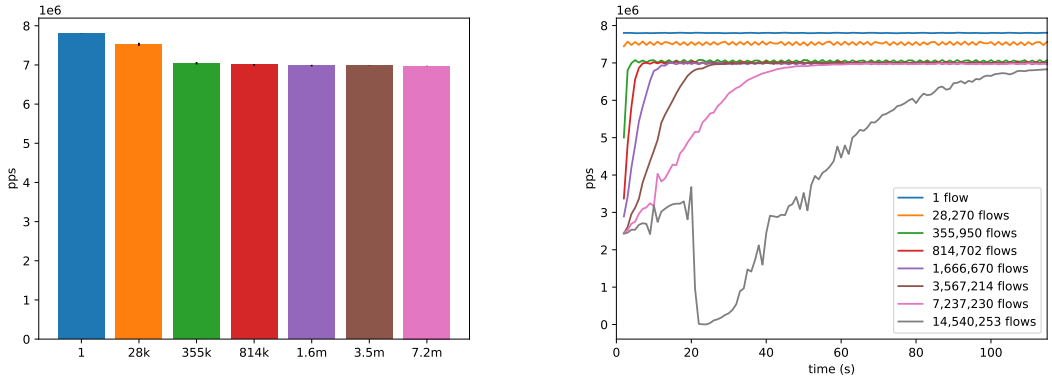


Figure 8.1: Testbed Topology



(a) Stable throughput

(b) Timeline showing reduced throughput due to initial flow handling overhead

Figure 8.2: NAT server pps with different flow counts

The Evaluation testbed topology is depicted in Figure 8.1. An Edge-core AS5812-54X switch, running PicOS 2.11.7 (based on Debian Wheezy), is used for running the switch NAT Implementation. The server NAT is running on a Host equipped with a six-core Intel Xeon E5-1650v3 CPU (3.5 GHz), 16GB RAM and a 4 Port Intel X710 10GbE NIC running CentOS 7.3 with Kernel 3.10. The server NAT host has an additional out-of-band, 1GbE, 0.3ms RTT, connection to the switch which is used as synchronization channel.

For throughput measurements, two servers with the same hardware as the server NAT host, were used. Packets were generated using the pktgen Linux Kernel Module [pktgen16] at 14.58 Mpps (IPv4, UDP) with a packet size of 64 bytes, if not stated otherwise. RX-rates were measured every 0.1s using a Netbricks based NF that counts and discards all traffic, if not mentioned otherwise.

Latency measurements were performed on a different host, which has an Intel XeonE5-1650v4 (3.6GHz), 32GB RAM and a Solarflare X2522 10GbE two port NIC, which can timestamp RX and TX packets in hardware with nanosecond precision.

8.1 Throughput

8.1.1 Server NAT

We first evaluate the throughput of the server NAT implementation without considering syncronization with the switch NAT. NetBricks [PHJ+16], reports a 8.52 Mpps single core throughput for their NAT implementation, FTMB [SGB+15] around 3 Mpps and StatelessNF [KCH+15] 3.5 Mpps. The NetBricks evaluation however does not state how many flows were used to achieve this number and for FTMB the used packet size is not mentioned. Our server NAT implementation achieves a throughput of $7.80 \times 10^6 \pm 4615$ pps for a single flow, internal to external. For external to internal traffic (with flow state established), the numbers are similar. For external traffic, if no flow state exists, the server NAT can drop $14.34 \times 10^6 \pm 27350$ pps.

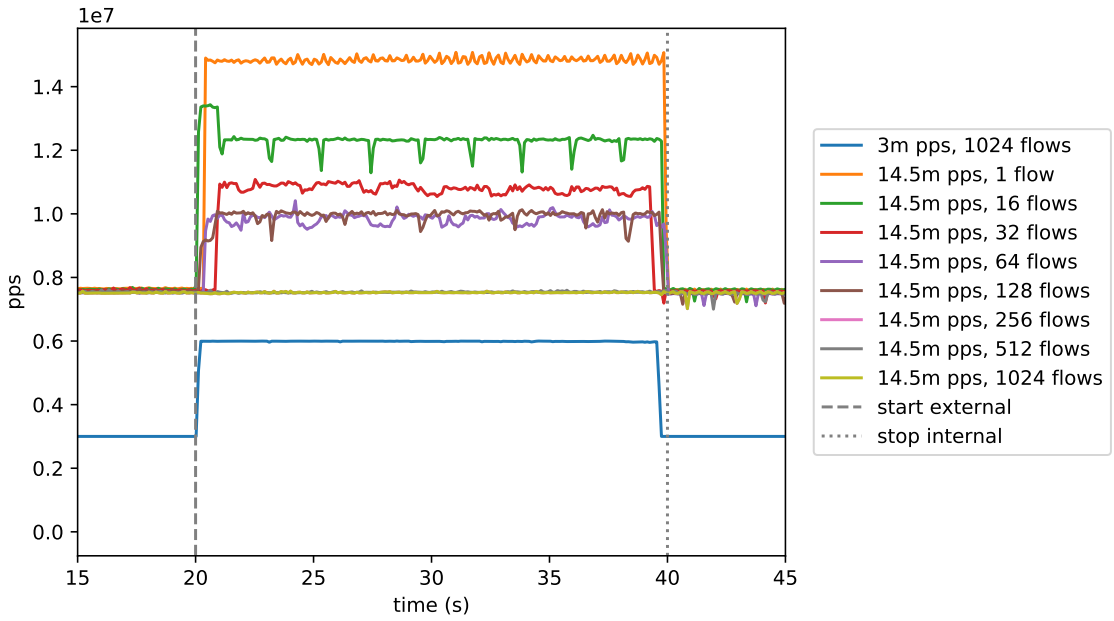


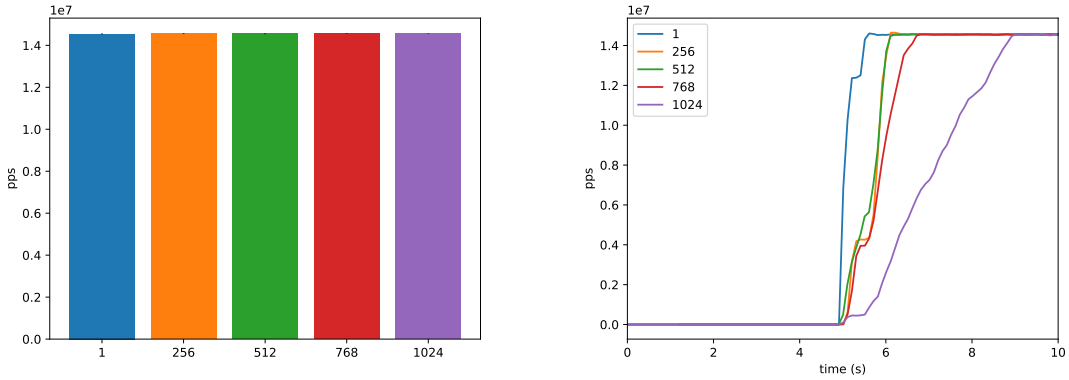
Figure 8.3: Effect of bidirectional traffic on throughput

Figure 8.2a shows how the throughput scales with the number of flows. For a single flow the throughput of $7.80 \times 10^6 \pm 4615$ pps is slightly higher than with 28 thousand flows at $7.53 \times 10^6 \pm 39531$ pps. For larger flow counts the throughput was stable around 7Mpps. Figure 8.2b shows that the initial, more involved, processing of new flows takes a toll on throughput until all flows have their state established. The overhead of new flow processing is discussed in more detail regarding latency in 8.2. For 14 million flows a massive dip in throughput is observed, before it recovers and stabilizes around 7Mpps. While figure 8.2b only shows a single run, this behavior was reliably reproducible. The source of this behavior is not clear, the implementation allocates all state data structures with enough size to fit the given NAT address pool, so no resizing should happen during runtime, which could be a possible explanation for this behavior.

Figure 8.3 shows the throughput under bidirectional traffic. Matching flows are generated on the internal and external hosts. The internal packet generator is started before the server NAT application, the external generator at 20 seconds, and at 40 seconds the internal generator is stopped. Because the NIC used by the packet generators can not be used to reliably measure the RX rate at the same time, the numbers are from the NAT server itself. While for a single flow the throughput doubles as one would expect, the combined throughput for more flows decreases rapidly and remains the same for flow counts of 256 and above. At a reduced throughput of 3 Mpps, the throughput is again doubled as expected. These results indicate that there must be a, bottleneck that limits the bidirectional throughput, possibly memory bandwidth as our test system is a single socket system.

8.1.2 Switch NAT

Next we will look at the switch NAT in isolation similar to the server NAT. For a single flow, the switch NAT, as expected, reaches a throughput of $14.54 \times 10^6 \pm 11082$ pps, which corresponds to the packet generation rate. The switch was able to process packets at line rate for up to 1024 Flows



(a) Stable throughput

(b) Timeline showing reduced throughput due to initial flow handling overhead

Figure 8.4: NAT switch pps with different flow counts

(uni and bidirectional), which corresponds with the limit of 2048 NAT rules (two rules per flow) in the PicOS documentation [posnat18]. For more flows, the switch showed erratic behavior (see 8.5) and had to be rebooted.

Figure 8.4b shows a similar timeline to figure 8.2b for the switch NAT. Notice that for 1024 flows, it already takes about 4 seconds to reach the full throughput. This indicates that the initial flow handling on the switch is slower than on the server. Again, this will be discussed in more detail regarding latency in 8.2.

8.1.3 Split processing

The new flow processing overhead on the switch is about 2 magnitudes higher (see 8.2) than on the server. Using asynchronous state transfer in combination with the split processing mode, the server NAT can handle packet processing for new flows and the switch will take over the processing after receiving the state from the server. Figure 8.5 shows the observed throughput on the server NAT and the receiving host for 1024 flows.

8.2 Latency

Next we will look at the latency overhead introduced by the switch and server based NAT implementations. Figure 8.6 shows that the server NAT implementation is an order magnitude slower than the on switch processing. While the switch NAT shows, as expected, a stable latency distribution, the server NAT shows high tail latency of up to 10^4 ns. For established flows, the synchronization and new flow handling modes have no impact on latency. Figure 8.8a shows the same data as cumulative distribution function.

For packets that have no state associated to their flow, the server implementation, at around 10^4 ns latency, is two magnitudes faster than the switch implementation at 10^6 ns. However, both implementations show tail latency of up to (server) or over (switch) 10^8 ns, as shown in Figure

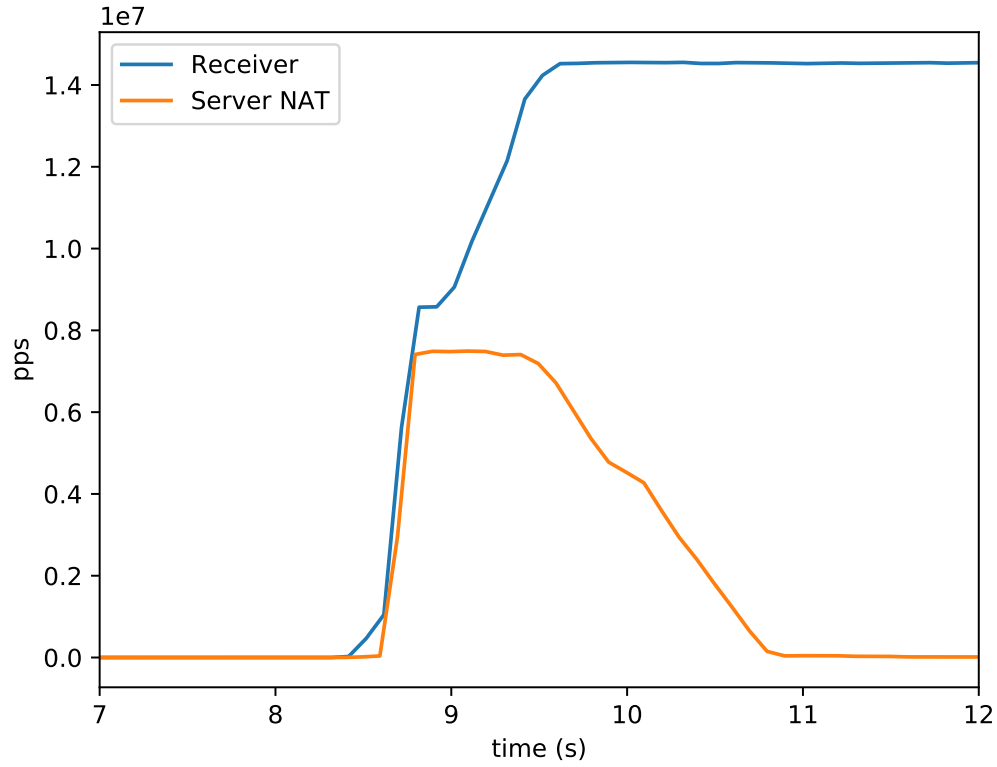


Figure 8.5: Split processing throughput at the server NAT and the receiver host, 1024 flows.

8.7. The box plot for the *sync server* processing mode demonstrates the impact of the synchronous synchronization mode: The server has to wait for the acknowledgment from the switch, pulling the processing latency to the switch level. Figure 8.8b shows the same data as cumulative distribution function.

8.3 Failover

To evaluate the effect of a failover event from the server NAT to the switch NAT on the packet stream, we first take a look at what (approximately) happens during a failover event:

1. Normal operation, state is shared between server and switch NAT
2. Server NAT crashes, $t = 0$
3. Kernel terminates the servers sync channel TCP socket and sends a `FIN+ACK`
4. Switch NAT receives the `FIN+ACK` after $RTT_{sync}/2 \approx$ **0.15 ms**

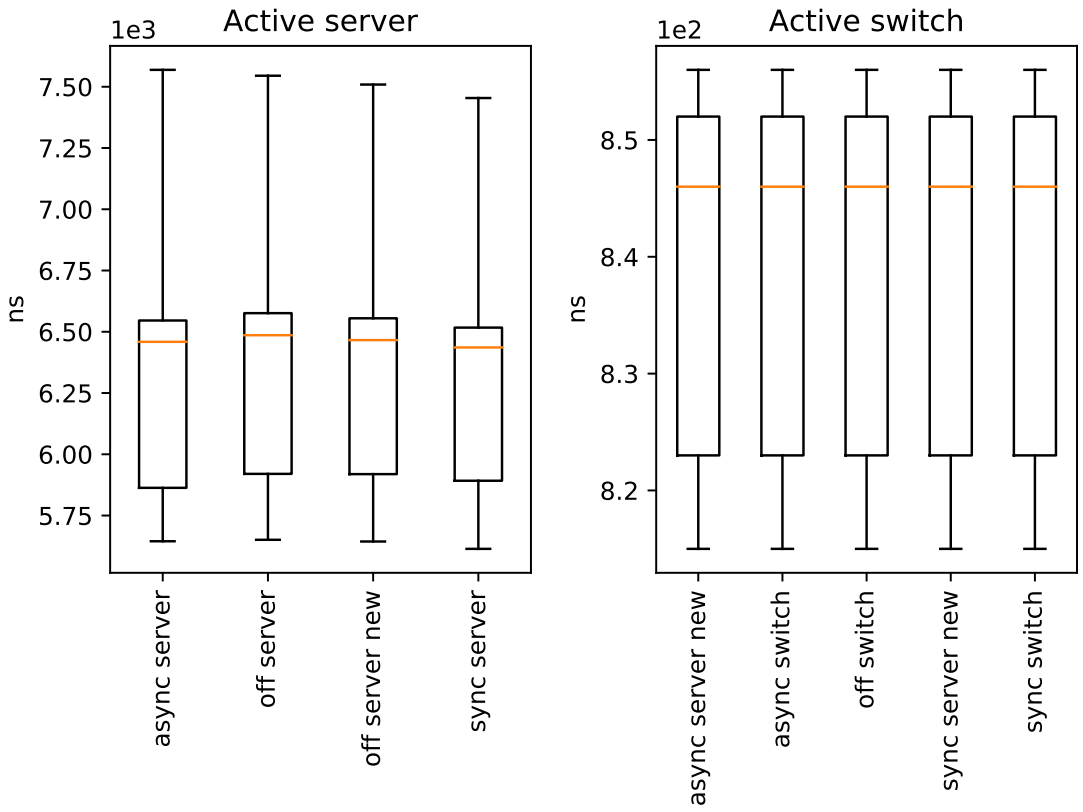


Figure 8.6: Per packet latency for established flows, 60 bytes, 500 pps, outliers not shown (cf. Figure 8.8a)

5. Switch NAT issues `FLOW_MOD` and a `BARRIER_REQUEST` OpenFlow messages ≈ **0.28 ms**[1]
6. The switch processes the `FLOW_MOD` messages and deletes the forwarding entries to the server NAT and responds with a `BARRIER_REPLY` OpenFlow message ≈ **0.28 ms**[2]
7. Packets is now handled by the switch data plane

In a simple test setup with two hosts with synchronized clocks (NTP), the delay from sending a SIGTERM to a process on the one host to receiving the socket error on the other host had a mean delay of 0.36 ±0.05 ms, with a max of 0.46 ms.

So even if we assume a 1ms delay to account for eventual clock inaccuracies, the failover should take under 2ms to complete. To validate this hypothesis, we send packets with a rate of 1 packet per ms and count the dropped packets during a failover event. Figure 8.9a shows a histogram of the results for 25 trials with a mean of 13.32 packets dropped during failover, or a drop window of

[1] Mean of data in Figure 8.9c
[2] Mean of data in Figure 8.9d

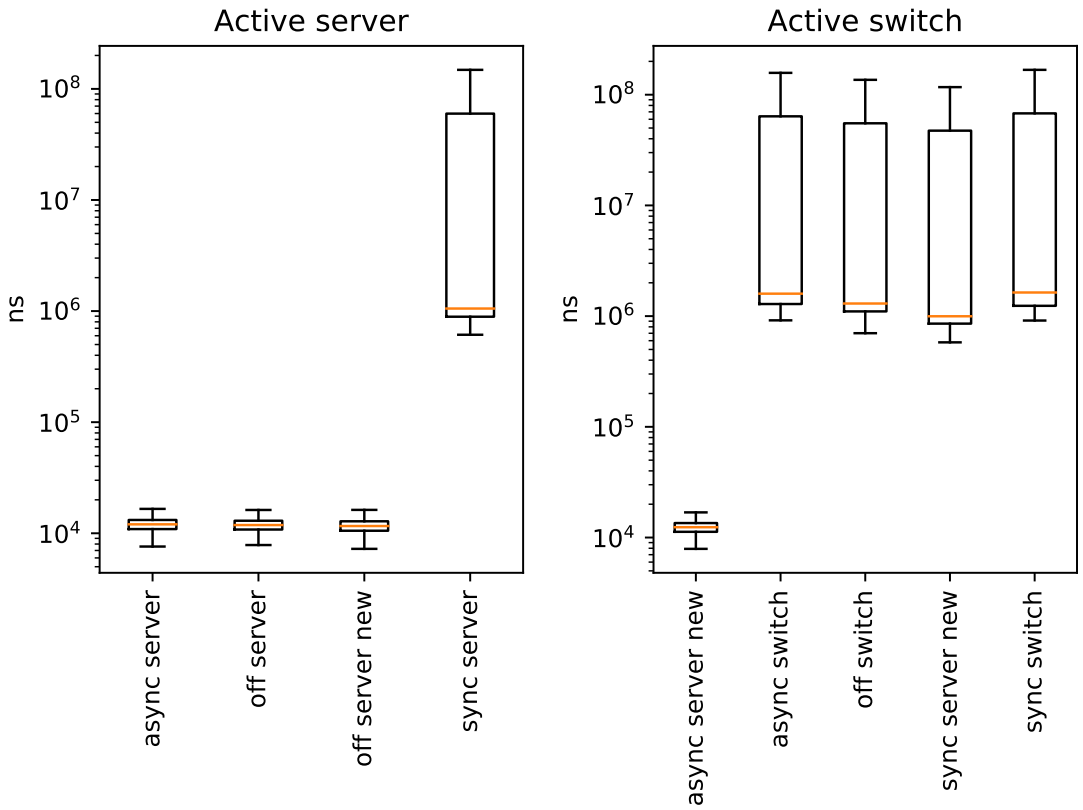
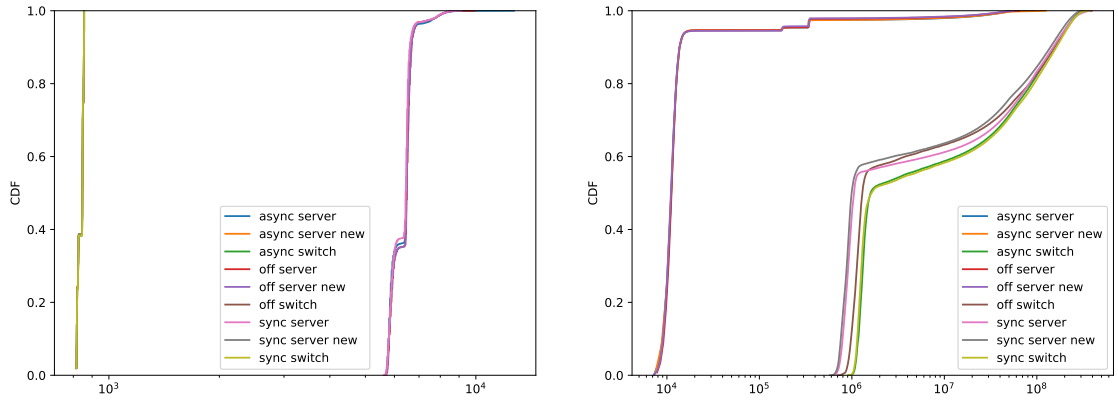


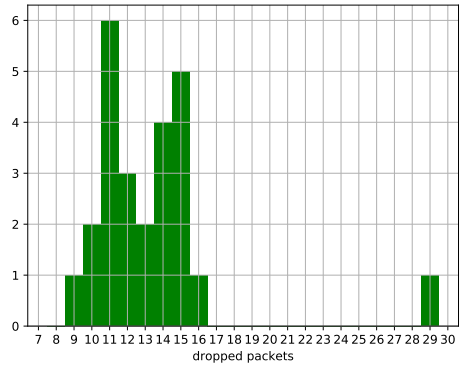
Figure 8.7: Per packet latency for new flows, 60 bytes, 500 pps, outliers not shown (cf. Figure 8.8b)



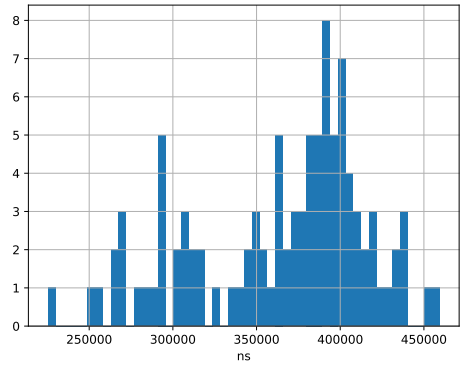
(a) established flows

(b) new flows

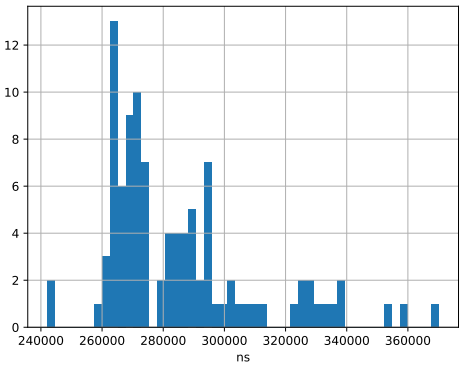
Figure 8.8: CDF of per packet latency



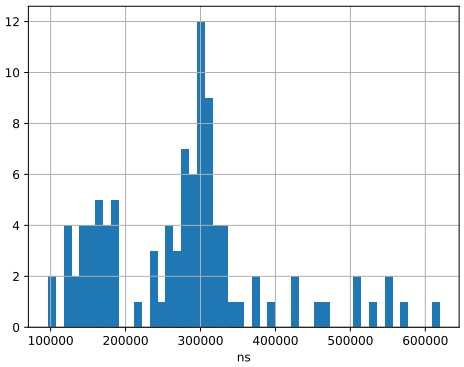
(a) Histogram showing the number of dropped packets during a failover event at 1000 pps, n=25, single flow



(b) Duration from process termination on one host to a socket error on another host, n=100



(c) Duration from detection of a disconnect to sending of the barrier request after the flow mods that delete the forwarding rules to the server NAT, n=100



(d) Duration from barrier request sent out to barrier reply received, n=100

Figure 8.9: Histograms showing the observed packet drops during a failover event (green) and the duration of different operations during failover (blue)

about 13ms. Even when assuming the worst case data points for detection to reaction of 0.37 ms, see Figure 8.9c, and a barrier response time of 0.62 ms, see Figure 8.9d and the max. delay of the Kernel (0.46 ms), the result is still below 2 ms, and far from the measured drop window.

Figure 8.10 shows that the drop window scales with the flow count. The detection to reaction duration however does not and, more interestingly, the barrier request to reply duration does neither. [KPKC18] shows that switch implementations are not always compliant to the OpenFlow specification, especially that flow modifications can take several hundred milliseconds to be effective after the switch sent a `BARRIER_REPLY` and that update latency gets worse with increasing flow count.

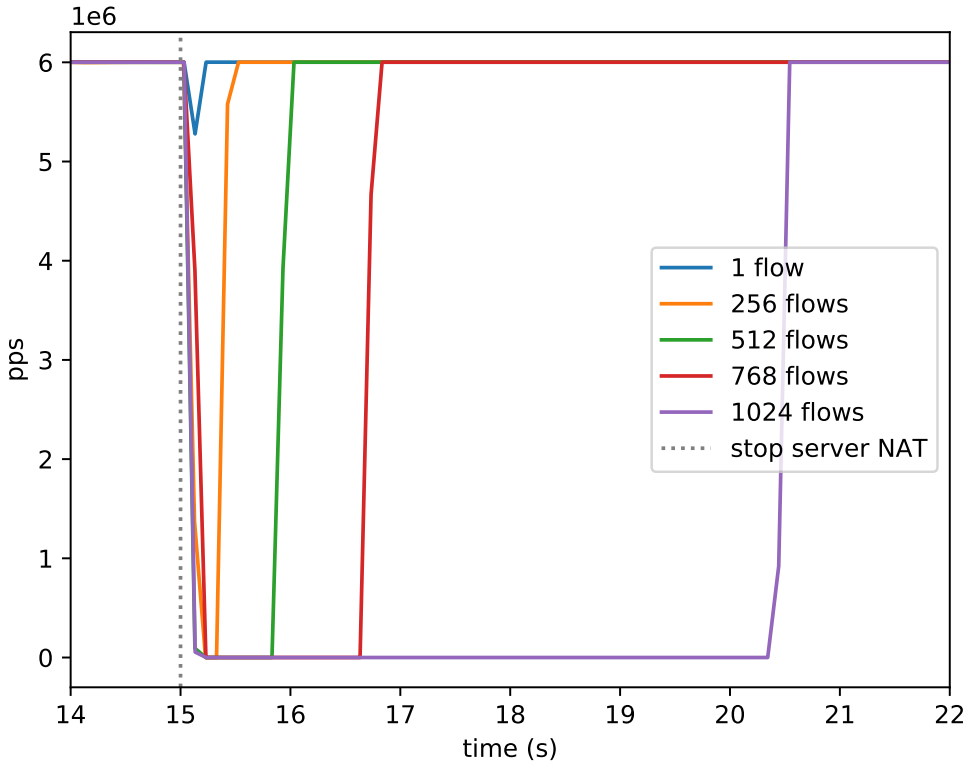


Figure 8.10: Timeline of failover events showing the packet rate at the receiver for different flow counts, 6 Mpps

8.4 Server NAT Breakdown

Figure 8.11 shows how long the individual processing steps of a packet in the server NAT implementation take. The numbers were measured with the (micro-) benchmarking tooling in the Rust test framework, see [rustbench18; rustbenchsrc18] for details. The dashed line marks 67.2 ns, the time that is available to sequentially process a single packet at 14.88 Mpps [10gbecalc14]. While processing for known flows seems to be close to that limit, the measurements only account for the core processing steps in isolation. Additional overhead from Netbricks, DPDK and other factors like lock acquisition, that are done once per packet batch, are not included. One might estimate that these would sum to around 60 ns, when calculating with the 7.80 Mpps single flow performance, but no actual measurements were performed to validate that number. For new flows, the additional processing time accounts for 72 ± 32 ns. These numbers differ form the measured packet latency for the server side processing with a mean of 10^4 ns. Batching and buffering in the RX/TX ring buffers before being actually handled are the likely explanation for this, plus two times the switch hop latency of around 840 ns.

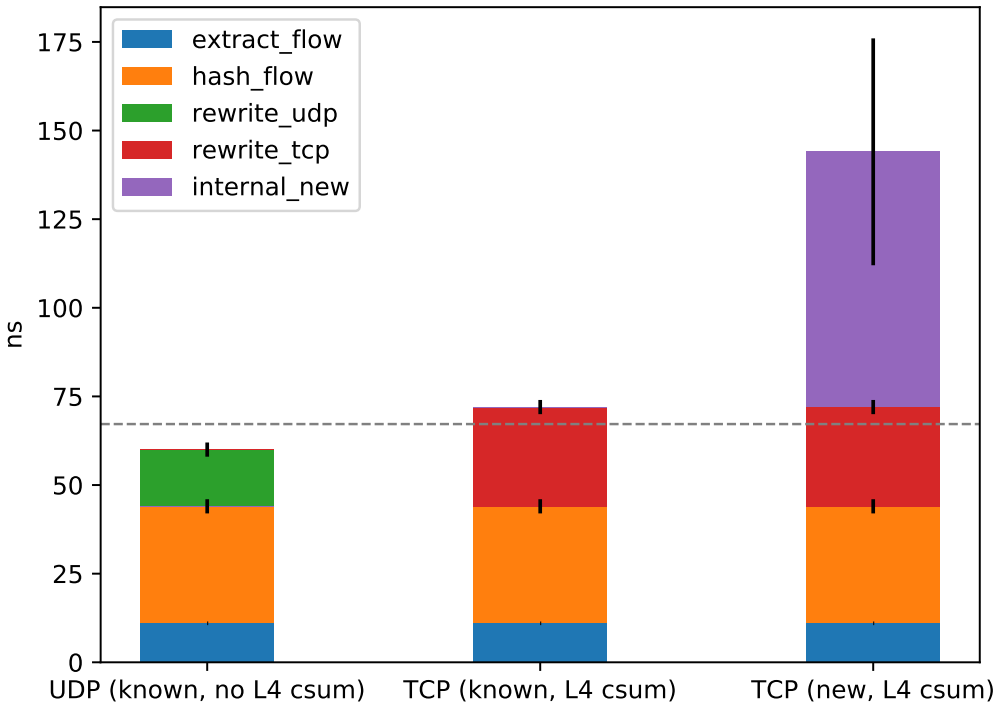


Figure 8.11: Duration of server NAT processing steps

8.5 Discussion

The evaluation shows that the server NAT implementation comes close (91.55%) to the reported throughput numbers of NetBricks, taking the more complex processing into account, this is a reasonable outcome. The server implementation also scales well regarding flow count for known flows.

New flow processing however shows extreme tail latency. One reason for this behavior could be that the implementation is not optimized for the new flow case, but rather for the common case of dealing with known flows. One major contributor to this tail latency is probably the lock on the external state map, which has to be acquired for every new flow, leaving room for possible improvements.

Another interesting phenomenon is that bidirectional traffic does not behave as expected when using non trivial flow counts. A possible explanation could be, that the shared cache of the (single socket) CPU is exhausted, limiting the throughput with non trivial state sizes by forcing access to the (comparatively) slow RAM. By using a system with independent sockets, this assumption could be validated.

The switch implementation can handle up to 1024 known flows at full (bidirectional) line rate, but is slower than the server implementation for new packets. For larger flow counts, the switch started to show nondeterministic behavior and, in some cases, while still responding to OpenFlow messages, failed to update the dataplane in response to these messages. During the failover evaluation, the switch failed to forward packets after deleting the forward-to-server rules, if the switch was not rebooted prior to each trial and, aligning with results of Kuźniar et al. [KPKC18] and Oudin et al. [OAR+18][3], the failover drop window scales with the flow count and barrier replies are not consistent with the observed data plane behavior.

[3] Same switch model and control plane OS as in our evaluation.

9 Conclusion and Future Work

This thesis gave an extensive overview on work related to NFV challenges like placement, chaining, state management and implementation. Based on the discussed literature, a simple, unified system model and design were proposed. A catalog of patterns that offer abstract solutions for NFV challenges and requirements was compiled. The patterns are based on approaches from literature as well as own concepts and fit in the presented system model. A prototype that applies a modest subset of the patterns to a common NF was implemented and evaluated. The evaluation showed that the implemented prototype offered good performance characteristics when compared to results from related work and that the implemented failover solution works as expected, when ignoring switch behavior that goes against the OpenFlow specification.

9.1 Future Work

Possible directions for future work could include implementing of a larger subset of the presented patterns and overall system, for example the overall orchestration or chaining mechanism. This could help to evaluate the proposed design and identify incorrect assumptions, missing details and potentially lead to the discovery of additional patterns.

It also seems worthwhile to explore the potential to distribute and delegate orchestration tasks in NFV workloads outside of the presented switch offloading patterns. Many scaling, fault tolerance or load balancing events only concern a limited context (topology, flowspace) and could be delegated to a local orchestrator to improve response times and reduce the overall load on the global orchestrator.

Another direction could be to investigate how NFV systems could be made accessible by a cloud provider to third party tenants that require strict isolation and compliance to SLAs.

Debugging and monitoring of NFV deployments could also be an interesting topic. While most of the components already collect metrics that are used for e.g. load balancing, offering an integrated, global view of the system could help to detect bottlenecks or errors in the forwarding, processing or configuration, which would be a valuable capability.

In the context of hardware acceleration, the requirement to implement the same network function multiple times with vastly different abstractions can easily introduce subtle bugs and drastically increase the maintenance cost of network functions, an effect that was noticeable during the implementation of the presented distributed NAT NF. A library/system that offers a common abstraction for the different hardware contexts, without sacrificing the performance benefits (cf. machine learning frameworks like TensorFlow), would greatly improve the ergonomics of VNF development.

Bibliography

[10gbecalc14] J. Dangaard Brouer. *The calculations: 10Gbit/s wirespeed*. 2014. URL: http://netoptimizer.blogspot.com/2014/05/the-calculations-10gbits-wirespeed.html (cit. on p. 91).

[ABFL15] B. Anwer, T. Benson, N. Feamster, D. Levin. "Programming slick network functions". In: *Proceedings of the 1st acm sigcomm symposium on software defined networking research*. ACM. 2015, p. 14 (cit. on pp. 20, 22, 46).

[ABK+12] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, A. Vahdat. "xOMB: extensible open middleboxes with commodity servers". In: *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM. 2012, pp. 49–60 (cit. on p. 23).

[ACM+16] A. Alim, R. Clegg, L. Mai, L. Rupprecht, E. Seckler, P. Costa, P. Pietzuch, A. L. Wolf, N. Sultana, J. Crowcroft, A. Madhavapeddy, A. W. Moore, R. Mortier, M. Koleini, L. Oviedo, D. McAuley, M. Migliavacca. "FLICK: Developing and Running Application-Specific Network Services". In: *2016 USENIX Annual Technical Conference (ATC)*. USENIX. Denver, CO, USA: USENIX, 2016 (cit. on p. 23).

[BAM10] T. Benson, A. Akella, D. A. Maltz. "Network traffic characteristics of data centers in the wild". In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM. 2010, pp. 267–280 (cit. on p. 66).

[BDG+14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95 (cit. on pp. 15, 29).

[BHH16] A. Bremler-Barr, Y. Harchol, D. Hay. "OpenBox: a software-defined framework for developing, deploying, and managing network functions". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM. 2016, pp. 511–524 (cit. on pp. 21, 22, 24, 70).

[bincode18] T. Overby. *Bincode*. 2018. URL: https://github.com/TyOverby/bincode (cit. on p. 78).

[BRXM15] Z. Bronstein, E. Roch, J. Xia, A. Molkho. "Uniform handling and abstraction of NFV hardware accelerators". In: *IEEE Network* 29.3 (2015), pp. 22–29 (cit. on p. 28).

[BZ16] R. Bauer, M. Zitterbart. "Port based capacity extensions (pbces): Improving sdns flow table scalability". In: *Teletraffic Congress (ITC 28), 2016 28th International*. Vol. 1. IEEE. 2016, pp. 225–233 (cit. on pp. 29, 41).

[CLNR15] R. Cohen, L. Lewin-Eytan, J. S. Naor, D. Raz. "Near optimal placement of virtual network functions". In: *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE. 2015, pp. 1346–1354 (cit. on p. 20).

[dpdk18] DPDK Project. *DPDK*. 2018. URL: https://www.dpdk.org/ (cit. on pp. 16, 72).

[EYC+16] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, J. D. Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer." In: *NSDI*. 2016, pp. 523–535 (cit. on p. 41).

[FCS+14] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, 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)*. Seattle, WA: USENIX Association, 2014, pp. 543–546. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/fayazbakhsh (cit. on pp. 21, 40, 42, 49).

[GA15] A. Gember-Jacobson, A. Akella. "Improving the safety, scalability, and efficiency of network function state transfers". In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM. 2015, pp. 43–48 (cit. on p. 26).

[GGA+12] A. Gember, R. Grandl, A. Anand, T. Benson, A. Akella. "Stratos: Virtual middleboxes as first-class entities". In: *UW-Madison TR1771* 15 (2012) (cit. on pp. 20, 24, 46, 64).

[GJM+17] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, K. Park. "APUNet: Revitalizing GPU as Packet Processing Accelerator". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 83–96. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go (cit. on pp. 28, 29).

[GVP+14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, A. Akella. "OpenNF: Enabling innovation in network function control". In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 163–174 (cit. on pp. 25, 37, 38, 50, 51, 57–59).

[HJP+15] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, S. Ratnasamy. "Softnic: A software nic to augment hardware". In: *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155* (2015) (cit. on p. 22).

[HJPM11] S. Han, K. Jang, K. Park, S. Moon. "PacketShader: a GPU-accelerated software router". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 195–206 (cit. on pp. 28, 29).

[JHH+11] K. Jang, S. Han, S. Han, S. B. Moon, K. Park. "SSLShader: Cheap SSL Acceleration with Commodity Processors." In: *NSDI*. 2011 (cit. on p. 29).

[JMK+17] M. A. Jamshed, Y. Moon, D. Kim, D. Han, K. Park. "mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes." In: *NSDI*. 2017, pp. 113–129 (cit. on pp. 23, 34, 67).

[JWJ+14] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, K. Park. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems." In: *NSDI*. Vol. 14. 2014, pp. 489–502 (cit. on pp. 16, 23).

[KCH+15] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, E. Keller. "Stateless network functions". In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM. 2015, pp. 49–54 (cit. on pp. 27, 37, 38, 56, 84).

[KDBR17] T. Kohler, F. Dürr, C. Bäumlisberger, K. Rothermel. "InFEP—Lightweight virtualization of distributed control on white-box networking hardware". In: *Network and Service Management (CNSM), 2017 13th International Conference on*. IEEE. 2017, pp. 1–6 (cit. on pp. 29, 31, 42).

[KDR17] T. Kohler, F. Dürr, K. Rothermel. "ZeroSDN: A Highly Flexible and Modular Architecture for Full-range Network Control Distribution". In: *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. ANCS '17. Beijing, China: IEEE Press, 2017, pp. 25–37. ISBN: 978-1-5090-6386-4. DOI: 10.1109/ANCS.2017.13. URL: https://doi.org/10.1109/ANCS.2017.13 (cit. on p. 29).

[KLL+97] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 654–663 (cit. on p. 41).

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297 (cit. on pp. 21, 24).

[KMD+18] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, K. Rothermel. "P4CEP: Towards In-Network Complex Event Processing". In: *Proceedings of the 2018 Morning Workshop on In-Network Computing*. NetCompute '18. Budapest, Hungary: ACM, 2018, pp. 33–38. ISBN: 978-1-4503-5908-5. DOI: 10.1145/3229591.3229593. URL: http://doi.acm.org/10.1145/3229591.3229593 (cit. on p. 29).

[KPKC18] M. Kuźniar, P. Perešíni, D. Kostić, M. Canini. "Methodology, Measurement and Analysis of Flow Table Update Characteristics in Hardware OpenFlow Switches". In: *Computer Networks* 136 (2018), pp. 22–36. DOI: doi:10.1016/j.comnet.2018.02.014 (cit. on pp. 90, 93).

[LC15] Y. Li, M. Chen. "Software-defined network function virtualization: A survey". In: *IEEE Access* 3 (2015), pp. 2542–2553 (cit. on p. 19).

[loci18] Project Floodlight. *LOCI*. 2018. URL: https://github.com/floodlight/loxigen-artifacts/tree/master/loci (cit. on p. 71).

[LQ16] X. Li, C. Qian. "A survey of network function placement". In: *Consumer Communications & Networking Conference (CCNC), 2016 13th IEEE Annual*. IEEE. 2016, pp. 948–953 (cit. on pp. 19, 20).

[lwn15] J. Corbet. *Improving Linux networking performance*. 2015. URL: https://lwn.net/Articles/629155/ (cit. on p. 16).

[MAR+14] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, F. Huici. "ClickOS and the art of network function virtualization". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association. 2014, pp. 459–473 (cit. on p. 24).

[MRA+14] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, A. L. Wolf. "NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres". In: *10th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM. Sydney, Australia: ACM, 2014 (cit. on p. 29).

[MSG+16] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, R. Boutaba. "Network function virtualization: State-of-the-art and research challenges". In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 236–262 (cit. on p. 19).

[OAR+18] R. Oudin, G. Antichi, C. Rotsos, A. W. Moore, S. Uhlig. "OFLOPS-SUME and the art of switch characterization". In: *IEEE Journal on Selected Areas in Communications* (2018) (cit. on p. 93).

[onap18] ONAP Authors. *ONAP*. 2018. URL: https://www.onap.org/ (cit. on p. 25).

[opnfv18] OPNFV Authors. *OPNFV*. 2018. URL: https://www.opnfv.org/ (cit. on p. 25).

[osm18] ETSI. *Open Source MANO*. 2018. URL: https://osm.etsi.org/ (cit. on p. 25).

[ovs18] The Open vSwitch Authors. *Open vSwitch*. 2018. URL: http://www.openvswitch.org/ (cit. on p. 72).

[PHJ+16] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, S. Shenker. "NetBricks: Taking the V out of NFV." In: *OSDI*. 2016, pp. 203–216 (cit. on pp. 22, 67, 71, 84).

[pktgen16] The Linux Foundation. *pktgen*. 2016. URL: https://wiki.linuxfoundation.org/networking/pktgen (cit. on p. 84).

[PLH+15] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, S. Shenker. "E2: a framework for NFV applications". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pp. 121–136 (cit. on pp. 20, 22, 24, 25, 34, 42, 49, 60, 63).

[posnat18] Pica8 Inc. *Configuring NAT flow - PicOS 2.11.7 Configuration Guide - Docs*. 2018. URL: https://docs.pica8.com/display/PicOS2117cg/Configuring+NAT+flow (cit. on p. 86).

[QTC+13] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, M. Yu. "SIMPLE-fying middlebox policy enforcement using SDN". In: *ACM SIGCOMM computer communication review*. Vol. 43. 4. ACM. 2013, pp. 27–38 (cit. on pp. 21, 49, 64).

[rfnv18] Servo Contributors. *Fowler–Noll–Vo hash function*. 2018. URL: https://github.com/servo/rust-fnv (cit. on p. 76).

[rhp15] A. Beingessner. *Rust hashing algorithm benchmarks*. 2015. URL: http://cglab.ca/~abeinges/blah/hash-rs/ (cit. on p. 76).

[Riz12] L. Rizzo. “Netmap: a novel framework for fast packet I/O”. In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 101–112 (cit. on p. 16).

[rust18] The Rust contributors. *The Rust Programming Language*. 2018. URL: https://lwn.net/Articles/629155/ (cit. on p. 71).

[rustbench18] The Rust Project Developers. *https://doc.rust-lang.org/unstable-book/library-features/test.html*. 2018. URL: http://netoptimizer.blogspot.com/2014/05/the-calculations-10gbits-wirespeed.html (cit. on p. 91).

[rustbenchsrc18] The Rust Project Developers. *rust/src/libtest/lib.rs*. 2018. URL: https://github.com/rust-lang/rust/blob/master/src/libtest/lib.rs#L1579-L1645 (cit. on p. 91).

[RWJ13] S. Rajagopalan, D. Williams, H. Jamjoom. “Pico replication: A high availability framework for middleboxes”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 1 (cit. on pp. 27, 37, 38, 54, 57).

[RWJW13] S. Rajagopalan, D. Williams, H. Jamjoom, A. Warfield. “Split/Merge: System Support for Elastic Execution in Virtual Middleboxes.” In: *NSDI*. Vol. 13. 2013, pp. 227–240 (cit. on pp. 27, 50, 59, 64).

[SAA+17] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, P. Kalnis. “In-Network Computation is a Dumb Idea Whose Time Has Come”. In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM. 2017, pp. 150–156 (cit. on p. 29).

[SER+12] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, G. Shi. “Design and implementation of a consolidated middlebox architecture”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 24–24 (cit. on pp. 20, 21, 70).

[serde18] The Rust contributors. *Serde*. 2018. URL: https://serde.rs/ (cit. on p. 78).

[SGB+15] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. “Rollback-recovery for middleboxes”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. ACM. 2015, pp. 227–240 (cit. on pp. 13, 28, 39, 53, 55, 84).

[SGG+17] *Emu: Rapid Prototyping of Networking Services*. Santa Clara, CA, USA: USENIX, 2017 (cit. on pp. 28, 29).

[SR12] J. Sherry, S. Ratnasamy. *A Survey of Enterprise Middlebox Deployments*. Tech. rep. UCB/EECS-2012-24. EECS Department, University of California, Berkeley, 2012. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-24.html (cit. on p. 13).

[TR98] D. G. Thaler, C. V. Ravishankar. “Using name-based mappings to increase hit rates”. In: *IEEE/ACM Transactions on networking* 6.1 (1998), pp. 1–14 (cit. on p. 41).

[TS06] A. S. Tanenbaum, M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275 (cit. on p. 52).

[TT05] J. S. Turner, D. E. Taylor. “Diversifying the internet”. In: *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE*. Vol. 2. IEEE. 2005, 6–pp (cit. on p. 17).

[TW10] A. S. Tanenbaum, D. J. Wetherall. *Computer Networks*. 5th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2010. ISBN: 978-0132126953 (cit. on pp. 16, 17).

[ZHR+16] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood. “Flurries: Countless fine-grained nfs for flexible per-flow customization”. In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM. 2016, pp. 3–17 (cit. on p. 23).

[ZLZ+16] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, T. Wood. “OpenNetVM: A platform for high performance network service chains”. In: *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*. ACM. 2016, pp. 26–31 (cit. on p. 23).

All links were last followed on August 30, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature