TECHNISCHE
UNIVERSITÄT
DARMSTADT

UTILIZING ADVANCED NETWORK CONTEXT TO OPTIMIZE SOFTWARE-DEFINED NETWORKS

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertationsschrift

von

MARC WERNER, M.SC.

Geboren am 27. Februar 1983 in Alzenau i. Ufr.

Erstreferent: Prof. Dr.-Ing. Matthias Hollick
Korreferent: Prof. Dr. Thomas Plagemann

Tag der Einreichung: 11. September 2017
Tag der Disputation: 10. November 2017

Darmstadt, 2017
Hochschulkennziffer D17

ABSTRACT

Legacy network systems and protocols are mostly static and keep state information in silo-style storage, thus making state migration, transformation and re-use difficult. Software-Defined Network (SDN) approaches in unison with Network Function Virtualization (NFV) allow for more flexibility, yet they are currently restricted to a limited set of state migration options. Additionally, existing systems and protocols are mostly tailored to meet the requirements of specific application scenarios. As a result, the protocols cannot easily be adapted to novel application demands, organically growing networks, etc.

Impeding the sharing of networking and system state, along with lacking support for dynamic transitions between systems and protocols, severely limits the ability to optimally manage resources and dynamically adapt to a desirable overall configuration. These limitations not only affect the network performance but also hinder the deployment of new and innovative protocols as a hard break is usually not feasible and thus full support for legacy systems is required.

On the one hand, we propose a generalized way to collect, store, *transform*, and share context between systems and protocols in both the legacy Internet as well as NFV/SDN-driven networks. This allows us to share state information between multiple systems and protocols from NFs over BGP routers to protocols on all layers of the network stack.

On the other hand, we introduce an architecture for designing modular protocols that are built with transition in mind. We argue that the modular design of systems and protocols can remove the key limitations of today's monolithic protocols and allow for a more dynamic network management.

First, we design and implement a *Storage and Transformation Engine for Advanced Networking context (STEAN)* which constitutes a shared context storage, making network state information available to other systems and protocols. Its pivotal feature is the ability to allow for state transformation as well as for persisting state to enable future re-use.

Second, we provide a *Blueprint for Switching Between Mechanisms* that serves as a framework and guideline for developers to standardize and ease the process of designing and implementing systems and protocols that support transitions as a first order principle.

By means of experimentation, we show that our architecture covers a diverse set of challenging use cases in legacy systems—such as Wireless Multihop Networks (WMNs)—as well as in NFV/SDN-enabled systems. In particular, we demonstrate the feasibility of our approach by migrating state information between two instances of the PRADS NF in a virtualized Mininet environment, and show that our solution outperforms state of the art frameworks that are specifically built for NF migration. We further demonstrate that a dynamic switch between WMN routing protocols is possible at runtime and that the state information can be reutilized for bootstrapping novel protocol modules, thus minimizing the control overhead.

ZUSAMMENFASSUNG

Existierende Netzwerksysteme und -protokolle sind meist statisch und halten Zustandsinformationen in einem silo-artigen Speicher, was die Migration, Transformation und Wiederverwendung dieser Zustände erschwert. Software-Definierte Netze (SDN) in Kombination mit der Virtualisierung von Netzwerkfunktionen (NFV) ermöglicht mehr Flexibilität, ist aber derzeit auf eine bestimmte Optionen zur Zustandsmigration beschränkt. Darüber hinaus sind vorhandene Systeme und Protokolle meist auf die Anforderungen von konkreten Anwendungsszenarien zugeschnitten, weshalb diese nicht einfach an neuartige oder geänderte Anforderungen sowie organisch wachsende Netzwerke angepasst werden können.

Diese Beschränkungen, sowie die fehlende Unterstützung für dynamische Übergänge zwischen Systemen und Protokollen, beeinträchtigen die Fähigkeit, Ressourcen optimal zu verwalten und sich dynamisch an eine wünschenswerte Gesamtkonfiguration anzupassen. Sie beeinflussen nicht nur die Leistung des Netzwerks, sondern behindern auch den Einsatz neuer und innovativer Verfahren, da eine Unterbrechung des Datenflusses während des Betriebs normalerweise nicht akzeptabel ist und daher neue Systeme die existierenden Protokolle vollständig unterstützen müssen.

Zum einen schlagen wir daher eine allgemeine Möglichkeit zum Sammeln, Speichern, Transformieren und Teilen von Zustandsinformation zwischen Systemen und Protokollen sowohl im Internet als auch in einer NFV/SDN-Umgebung vor. Dies ermöglicht es uns, Zustandsinformationen zwischen mehreren Systemen und Protokollen von NFs über BGP-Router zu Protokollen auf allen Ebenen des Netzwerkstapels zu teilen.

Zum anderen präsentieren wir eine Architektur für den Entwurf von modularen Protokollen, die Schaltvorgänge zur Laufzeit explizit unterstützt. Der modulare Entwurf von Systemen und Protokollen beseitigt die Beschränkungen der heutigen monolithischen Protokolle und ermöglicht ein dynamischeres Netzwerkmanagement.

Zunächst entwerfen und implementieren wir eine Speicher- und Transformations-Einheit für den erweiterten Netzwerkkontext (STEAN), die einen gemeinsamen Kontextspeicher bietet und Zustandsinformationen aus dem Netzwerk für andere Systeme und Protokolle zur Verfügung stellt. Die entscheidende Fähigkeit dieses Systems ist es, sowohl eine Transformation von Zustandsinformationen durchzuführen als auch diese Informationen dauerhaft zu speichern und für eine zukünftige Nutzung bereitzustellen.

Weiterhin bieten wir einen Bauplan für Mechanismen an, die eine explizite Unterstützung von Umschaltvorgängen zur Laufzeit bieten. Dieser Plan dient als Rahmenwerk und Leitfaden für Entwickler, um den Entwurf und die Implementierung von Systemen und Protokollen, die Übergänge als Kernprinzip unterstützen, zu standardisieren und zu vereinfachen.

Durch Experimente zeigen wir, dass unsere Architektur eine Vielzahl von anspruchsvollen Anwendungsfällen in bestehenden Systemen, wie Drahtlosen Multihop Netzwerken (WMNs), als auch in NFV/SDN-fähigen Systemen abdecken kann. Insbesondere präsentieren wir die Machbarkeit unseres Ansatzes durch die Migration von Zustandsinformationen zwischen zwei Instanzen der PRADS NF in einer virtualisierten Mininet-Umgebung und zeigen, dass unsere Lösung den Stand der Technik übertrifft, obwohl

diese Rahmenwerke speziell für die NF-Migration entwickelt wurden. Wir zeigen weiterhin, dass ein dynamischer Wechsel zwischen WMN Routingprotokollen zur Laufzeit möglich ist und dass bestehende Zustandsinformationen für das initiale Starten von neuartigen Protokollmodulen wiederverwendet werden können, wodurch die Menge der nötigen Kontroll- und Steuerungsnachrichten minimiert wird.

ACKNOWLEDGMENTS

Writting this thesis was a long and winding road, and would have not been possible without the support of a large group of people. There are those who accompanied me during the whole time and those who where there for just a short period but showed up at the right moment to give directions. Thus, a list of acknowledgements can never be complete. However, I would like to thank two groups of people without whom this thesis would not be possible at all:

The colleagues, students and fellow researchers that supported this work. Not only those who directly supported me on the topics discussed within this thesis, and those who shared their insights during fruitful discussions. But also those who were available for social activities, who opened my mind to different directions, and that became more than work colleagues over the course of the years.

My friends and family with whom I celebrated the successes but who also cheered me up during times of frustration and who always supported me no matter what happened in life. The people that give the joy in life and without whom it would have not been possible to even get to the stage of starting this thesis.

As research is not only about serious work but also about living a life that allows for creativity, I will close with the words of Steve Jobs who—eventhough not a classical researcher—influenced our disipline propably more than any other person in the last two decades: “Stay Hungry. Stay Foolish.”

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

5G network	fifth generation mobile network
AI	Artificial Intelligence
AODV	Ad hoc On-Demand Distance Vector Routing Protocol
API	Application Programming Interface
ARP	Address Resolution Protocol
AS	Autonomous System

BGP	Border Gateway Protocol
BS	Base Station
CDF	Cumulative Distribution Function
Click	Click Modular Router
D2D	device-to-device
DCN	Data Center Network
DMA	Direct Memory Access
DPI	Deep Packet Inspection
DSL	Domain Specific Language
eBGP	Exterior Border Gateway Protocol
FIB	Forwarding Information Base
HA	High Availability
I/O	Input/Output
iBGP	Interior Border Gateway Protocol
IDS	Intrusion Detection System
IoT	Internet of Things
IP	Internet Protocol
ISP	Internet Service Provider
IXP	Internet Exchange Point
LoC	Lines of Code
LRU	Least Recently Used
LVM	Lightweight Virtual Machine
MAC	Media Access Control
NAT	Network Address Translation
NF	Network Function
NFV	Network Function Virtualization
OLSR	Optimized Link State Routing Protocol
OS	Operating System

P2P	peer-to-peer
PCAP	Packet Capture
pps	packets per second
QoS	Quality of Service
RDBMS	Relational Database Management System
RIB	Routing Information Base
RTT	Round Trip Time
SAODV	Secure Ad hoc On-Demand Distance Vector Routing Protocol
SDN	Software-Defined Network
SLA	Service Level Agreement
SO	Shared Object
SOA	Service Oriented Architecture
SOLSR	Secure Optimized Link State Routing Protocol
STL	C++ Standard Library
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine
VoIP	Voice over IP
VRF	Virtual Routing and Forwarding
WAN	Wide Area Network
WMN	Wireless Multihop Network
XML	eXtensible Markup Language

PREVIOUSLY PUBLISHED MATERIAL

This thesis includes material previously published in peer-reviewed publications. In accordance with the regulations of the Computer Science department at TU Darmstadt, we list below the chapters which include verbatim fragments from these publications.

Chapter 1 and 2 include verbatim fragments from "STEAN: A Storage and Transformation Engine for Advanced Networking Context" by Marc Werner, Johannes Schwandke, Matthias Hollick, Oliver Hohlfeld, Torsten Zimmermann, Klaus Wehrle. In *Proceedings of the IFIP Networking Conference (IFIP Networking)*, 2016.

Chapter 3 and 4 build upon "A Blueprint for Switching Between Secure Routing Protocols in Wireless Multihop Networks" by Marc Werner, Jög Kaiser, Matthias Hollick, Elias Weingärtner and Klaus Wehrle. In: *Proceedings of the 14th International Symposium on a World of Wireless, Mobile and Multimedia Networks (IEEE WoWMoM, D-SPAN Workshop)*, 2013 as well as "STEAN: A Storage and Transformation Engine for Advanced Networking Context" by Marc Werner, Johannes Schwandke, Matthias Hollick, Oliver Hohlfeld, Torsten Zimmermann, Klaus Wehrle. In: *Proceedings of the IFIP Networking Conference (IFIP Networking)*, 2016.

Chapter 5 and 6 include verbatim text from "Mind the Gap – Understanding the Traffic Gap when Switching Communication Protocols " by Marc Werner, Tobias Lange, Matthias Hollick, Torsten Zimmermann, Klaus Wehrle. In *Proceedings of the 1st KuVS Workshop on Anticipatory Networks*, 2014. as well as "A Blueprint for Switching Between Secure Routing Protocols in Wireless Multihop Networks" by Marc Werner, Jög Kaiser, Matthias Hollick, Elias Weingärtner and Klaus Wehrle. In: *Proceedings of the 14th International Symposium on a World of Wireless, Mobile and Multimedia Networks (IEEE WoWMoM, D-SPAN Workshop)*, 2013. and "STEAN: A Storage and Transformation Engine for Advanced Networking Context" by Marc Werner, Johannes Schwandke, Matthias Hollick, Oliver Hohlfeld, Torsten Zimmermann, Klaus Wehrle. In: *Proceedings of the IFIP Networking Conference (IFIP Networking)*, 2016.

Appendix A includes verbatim text from "Mind the Gap – Understanding the Traffic Gap when Switching Communication Protocols " by Marc Werner, Tobias Lange, Matthias Hollick, Torsten Zimmermann, Klaus Wehrle. In *Proceedings of the 1st KuVS Workshop on Anticipatory Networks*, 2014.

1 INTRODUCTION

1.1 MOTIVATION

The design and management of communication networks currently undergoes massive changes towards realizing a more flexible management of complex networks. Recent efforts include rethinking the control plane design by applying Operating System (OS) design principles to realize Software-Defined Networks (SDNs). This triggered a shift in paradigm from a statically deployed infrastructure that operates within tight boundaries to an open system with well defined interfaces. It permits operators to dynamically adapt the network topology to changes in load, user demands or to environmental conditions without the need to physically replace components or completely redeploy the network.

Additionally, SDN enabled networks allow operators to converge existing infrastructure into one unified architecture and gain centralized control over the complete network management. Current deployments include carrier networks such as Google B4 [33] that connects datacenters across the globe with a highly available dynamic Wide Area Network (WAN).

A second innovation in the field of network operations in recent years is the introduction of Network Function Virtualization (NFV) which is inspired by the success of virtualization in the server market. Existing Network Functions (NFs) are designed and implemented as black boxes that built upon proprietary software and specialized hardware. This limits the deployment to a small predefined set of possible configurations and forces operators into a vendor lock-in.

The introduction of virtualization in the field of NFs allows operators to run their middleboxes on commodity hardware alongside other applications and supports the operation of NFs from different vendors on the same infrastructure. Additionally, the introduction of NFV enables the dynamic scaling of middleboxes to satisfy the demand of varying traffic patterns and load scenarios. The flexible instantiation of NFs helps to fulfill even tight Service Level Agreements (SLAs) without the need for over-provisioning.

The advances in both network management using SDN and traffic engineering by employing NFV aim at a more flexible and dynamic service deployment, increased resource utilization, improved energy efficiency, vendor independence, and ultimately decreased operational costs.

On the one hand, current research targets further improvements to the operation of these new services by proposing mechanisms for efficient scaling and dynamic rerouting of packets. This in turn requires the sharing of state collected within the NF between instances to prevent the service degradation or malfunction of middleboxes due to missing information.

This requirement has lead to the development of various systems that allow explicit state migration between NFs such as Split/Merge [57], OpenNF [28] or StatelessNF [34]. These solutions either require the developer of the NF to explicitly decouple the state from the implementation during development or try to identify and extract the state contained within existing NFs upon migration. The latter is done by adding annotations to the

source code or by replacing system calls with alternate implementations to intercept the communication with other systems.

On the other hand, there are systems and protocols that were never built with migration in mind. These protocols still form the large backbone of today's networks and, due to compatibility issues, will not be replaced in the near future. For example, the Border Gateway Protocol (BGP) was introduced with RFC 1105 [47] in 1989 and is still in active use today. The protocol constitutes the routing backbone of the Internet and thus takes a critical role in the network and the availability of the provided service.

However, the migration and replacement of BGP routers is a tedious task that requires long and accurate planning. In the absence of state sharing, router migration is challenging since reestablishing BGP routing sessions is expensive and network operation only allows for very short interruptions until the routing entries are invalidated [72]. Those interruptions would violate the SLAs usually in place at these critical points, and network operators therefore avoid a router migration whenever possible even if the complete network would benefit from this change. Even though some solutions for sharing state information between different BGP routers such as Router Grafting [37] or VROOM [73] exist, they are limited to specific operation environments, and require substantial extensions to the router implementation.

But not only protocols and systems in the network core are ponderous to change. Also access networks such as Wireless Multihop Networks (WMNs) are often running indefinitely without the possibility to alter the configuration or even deploy new protocols, thus limiting the options to adapt to changing user demands and environmental conditions. The applications of WMNs vary widely from access networks in urban environments [3] over connecting rural areas [1, 22] to networks that are only enabled in case of natural disasters or other emergencies when other infrastructure is not available [31, 52]. The requirements of these applications and the environment where these networks are deployed differ dramatically. Thus the networks currently can not easily be repurposed and a "one size fits all" approach is not feasible either.

Additionally, these networks are operated under a multitude of administrative domains and most devices are owned by end users that are neither able nor willing to constantly update their devices. Therefore, the once deployed protocols and systems often run for a long time without any possibility to alter the installation.

1.2 GOALS

Despite their success, current state sharing mechanisms are customized solutions tailored to specific use cases and are ignoring the fact that they continue to use closed *silo style* storage as shown in Figure 1.1a.

Our goal is to *break these silos open* and allow state to be shared within the complete network, ranging from SDN controllers over NFs to routers and protocol implementations—we call them *mechanisms* throughout this thesis—as shown in Figure 1.1b.

In this way, we enable legacy mechanisms to exchange state information with only minimal changes to the existing implementation and thus provide the possibility for better scaling and faster migration of these mechanisms. We also support the design and implementation of new mechanisms with a strict separation of functionality and state in mind. This clean segregation of operational information enables a wide sharing of state without the need for specialized solutions tailored to only a small number of use cases.

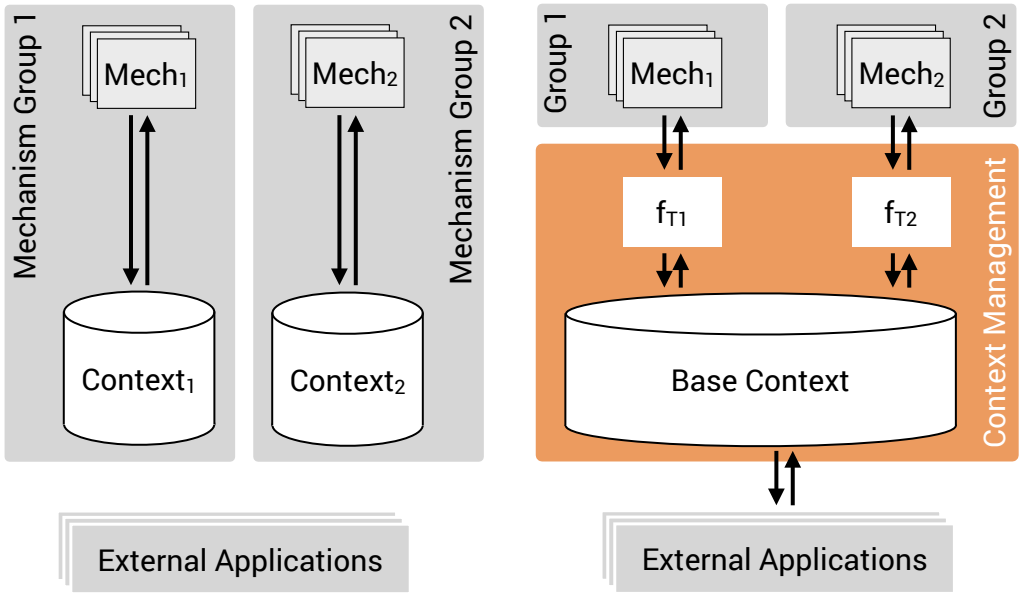

Figure 1.1: Current state sharing frameworks only allow sharing between functions of the same group (a). We enable sharing state between different groups by using a common representation and mapping the function specific state using the transformation functions f_{T1} and f_{T2} (b).

Additionally, we aim to extend the sharing of information *beyond* state information. Our goal is to establish the exchange of *network context*—an extended set of information introduced in detail in Section 2.1—to further open networks to a more flexible and dynamic management. Whenever we use the term *context* throughout this thesis, we refer to the network context.

In order to achieve our main goal of breaking the information silos open, we address the following sub goals in this thesis:

1.2.1 *Enable Native Context Sharing Across Mechanisms*

Our first goal is to provide a generalized way to share context in a diverse set of core functionality such as routing, in-network processing, and dynamic mechanism adaptation. This relies on collecting and managing context from different sources (e.g., NFs, BGP routers, protocols on all layers of the network stack) using their preferred state representation, thus replacing per-entity state storage with a shared context management. It makes this dynamic information available to other mechanisms, and stores and persists the current context to re-use at later points in time.

The system aids in seamlessly changing between mechanisms at runtime, or access persisted information describing the mechanism context from the past to reach fully operational state with minimal latency.

1.2.2 *Enable the Transformation of Context Information*

The global sharing of information would usually require all connected mechanisms to adapt their state representation to the format required by the context management system. However, this requires a major redesign of mechanisms, and thus hinders the introduction of a context management system.

Therefore, our second goal is to define transformation functions to re-use context in other mechanisms without agreeing on a common representation. These functions allow us to leave the internal state representation of the mechanisms unaltered while still sharing the context with other mechanisms.

Transformations enable context sharing between mechanisms that were not originally built with sharing in mind, and permits the seamless extension of existing mechanism stacks and network topologies. This adaptability enables the wide acceptance of a state sharing infrastructure as it significantly lowers the entry boundaries for participation.

1.2.3 *Enable Transition Support in Mechanisms*

Existing design principles for mechanisms do not take mechanism switches—also referred to as transitions [25]—into account. The existing solutions are either implemented as an supplementary addition to already existing designs or do not even incorporate the actual mechanisms but solely rely on external systems such as programmable networks to execute the switch.

Thus, our third goal is to introduce transition support as a first order design principle for mechanisms. The design, implementation and deployment of mechanisms with switching support further boosts the innovation in the Internet and supports the transition towards a flexible and dynamic network management.

1.3 CONTRIBUTIONS

Based on the goals of this thesis, we conceive, design, implement and evaluate the core components of an enhanced network architecture that natively supports the sharing, transformation and activation of context as well as enables the transition between the employed mechanisms. This architecture allows us to broadly extend the sharing and re-use of context information between mechanisms, and opens networks to allow for an even more flexible management.

On the one hand, we are able to extend the existing solutions for sharing state to provide more information to adjacent mechanisms and thus further improve their dynamic reconfiguration.

On the other hand, this approach allows to enable migration and scaling features for a large set of mechanisms that were not build with migration in mind.

In particular our contributions are as follows:

1.3.1 *Design and Implement a State Plane for Software-Defined Networks*

We design and implement the *Storage and Transformation Engine for Advanced Networking context (STEAN)*. It provides an architecture for a context management system that allows

us to share context between mechanisms in a diverse set of core functionality such as routing, in-network processing, and dynamic mechanism adaptation. It makes this dynamic information available to other mechanisms, and stores and persists the current context to re-use at later points in time. This way, STEAN enables any component in a network to access other components' context information, hence facilitating seamless network transitions.

The system aids in seamlessly changing between mechanisms at runtime, or access persisted information describing the mechanism context from the past to reach fully operational state with minimal latency. We demonstrate this by migrating NFs as described in the use case from Section 3.1.1, and by seamlessly switching between routing protocols in a WMN as discussed in Section 3.1.4.

1.3.2 *Provide an Architecture for Transformation Functions*

We define transformation functions to re-use context in other mechanisms without agreeing on a common representation. These functions enable context sharing between mechanisms that were not originally built with sharing in mind. Transformations allow the state plane to be integrated into legacy mechanisms and to interoperate with arbitrary mechanisms which permits the seamless extension of existing mechanism stacks and network topologies. Furthermore, transformation functions allow us to share context between different NFs that are—until now—only designed to exchange state between instances of the same implementation.

We demonstrate the feasibility of employing transformation functions and the performance of our implementation in both, a synthetic setup as well as the use cases utilized in Section 1.3.1. Our results show that employing transformations only imposes a small overhead on the operation of NFs and WMN routing protocols but provides the ability to share context between mechanisms that rely on a specialized state representation.

1.3.3 *Offer an Architectural Model for Switching Mechanisms*

We design a system for switching mechanisms in existing legacy systems. Our approach is based on a central controller that coordinates the transition and ensures that all connected instances change to the same target mechanism. The system not only supports the switch between mechanisms, but also allows for a parallel operation of multiple mechanisms to enable the reestablishment of information prior to carrying data traffic. This approach eliminates the traffic gap when switching and enables seamless transitions between mechanisms.

We implement a prototype of the architecture and evaluate our design based on the use case described in detail in Section 3.1.4. The implementation is based on the Click Modular Router [39] and extends existing WMN routing protocols to support our architecture. We show that our approach efficiently supports a seamless transition between mechanisms and thus enables the dynamic reconfiguration of legacy systems.

1.4 OUTLINE OF THE THESIS

The remainder of this thesis is structured as follows:

In Chapter 2, BACKGROUND AND RELATED WORK, we introduce the concept of network context along with a survey of the related work.

On the one hand, the survey focuses on work in the areas of network architecture and cross layer systems and specifically how state sharing can support new concepts and paradigms for designing and deploying network stacks or even complete networks.

On the other hand, we analyze the existing work in the area of router migration and network function virtualization, and how state management enables seamless migration and failover on those critical mechanisms.

We then introduce the use cases referenced throughout this thesis in Chapter 3, PROBLEM STATEMENT. The use cases include scenarios from recent developments in the field of NFV such as the seamless migration of middleboxes and the exchange of state between different types of NFs that allows for dynamic reconfiguration. Additionally, we also cover the sharing of information between legacy mechanisms such as BGP routers and WMNs. The chapter also introduces the challenges these use cases impose on the proposed solutions.

Our SYSTEM ARCHITECTURE is presented in Chapter 4. First, we present our work on the Storage and Transformation Engine for Advanced Networking context (STEAN) [75] which serves as a reference architecture on how to design a context management system in a modular and extensible manner. STEAN not only manages and stores context information from various mechanisms but also provides transformation functions that allow us to exchange context between different mechanisms that were never built with sharing in mind. We provide a proof-of-concept of our architecture along with a system analysis of our prototype.

Second, we introduce our blueprint for switching between mechanisms [74] that provides an architectural model for designing transition aware mechanisms as well as a prototypical implementation of the introduced concepts based on multiple WMN routing protocols using the Click Modular Router (Click). We also implemented a command and control environment that allows us to trigger and coordinate the switch between mechanisms across multiple nodes in a network segment.

In Chapter 5, we present an extensive EVALUATION of our architecture using two dedicated use cases. We show the system behavior while migrating network functions in Section 5.1. The experiments are conducted using a virtual environment based on Mininet, and show how the sharing of context information improves the migration time between two NF instances compared to state of the art frameworks such as OpenNF. We then evaluate the influence of sharing context information when switching routing protocols in Section 5.2 We show that a traffic gap always exists when switching mechanisms even on less utilized networks, and demonstrate how a context management system supporting transformation functions along with a modular mechanism design can eliminate this gap.

Finally, we conclude our work in Chapter 6. We discuss possible improvements to our prototype as well as a possible migration path towards a state plane. Additionally, we give an outlook on possible research directions and offer pointers for future work.

2 BACKGROUND AND RELATED WORK

Traditionally, networked systems are not designed with a clear separation of functionality and state information in mind. They are carefully engineered to provide the maximum forwarding performance and their architecture often sacrifices all other design goals such as extensibility towards faster packet processing. These mechanisms hold various state information in different parts of their architecture without providing a consistent or even convenient way to extract or modify this data.

Various solutions to share, migrate or replicate state information in such closed mechanisms exist. One of the key challenges is to identify the relevant data structures within the implementation as not all state information is of importance when sharing the data between instances. Table 2.1 gives an overview of the different types of state information available in networked systems.

	TYPE		VISIBILITY		SYNC FREQUENCY		
	STATIC	DYNAMIC	LOCAL	GLOBAL	NEVER	LOW	HIGH
Network Packets		x	x		x		
Flow Information		x		x			x
Configuration	x			x		x	
Caches		x	x		x		
Timers		x		x			x
Statistics		x		x	x		
Background Processes		x	x		x		

Table 2.1: Types of state information (adapted from [56, 57]).

Another challenge is to extract the relevant state from the current instance and transfer the information to another instance, thus creating an operational copy of the service. This is either done by accessing the structures in memory directly or by replacing the functions to access memory with customized versions that intercept the requests.

This thesis contributes to the area of state management for NFV as well as SDN. Additionally, our use cases described in Section 3.1 also include the context management of routing protocols in the Internet core and in WMNs. In the following, we therefore survey the related work in these areas and show the advantages of the presented approaches as well as their shortcomings.

Before we survey the related work, we introduce the concept of network context that is used throughout this thesis to describe the information handled by the presented systems.

2.1 NETWORK CONTEXT

Today, networked systems hold a large amount of state information from mechanisms as well as from applications that access the network and analyze or manipulate network traffic. This state information is mechanism specific, and often only an implicit description—expressed as data structures—exists within the used implementation. This state information is usually lost when an application is terminated or the mechanism stack is restarted.

Additionally, the existing systems only hold the current operational state and do not preserve or store historical information beyond the information needed for the current operation. The only structure that provides a history of operation is logging information generated by the mechanisms. However, application logs are usually unstructured and the data presented is selected based on the needs of a human operator. It is not meant to regenerate the operational state but to detect and analyze false behavior.

The term *state* is usually used to describe internal information, and *context* refers to all external factors that impact the behavior of a mechanism. We unify those two information sets and extend the included information.

Our goal is to not only share the current state of mechanisms, but to extend the shared information *beyond* the operational parameters. We therefore define the *network context*—shortly referred to as *context* throughout this thesis—as the extension of the state information and environmental parameters as shown in Figure 2.1.

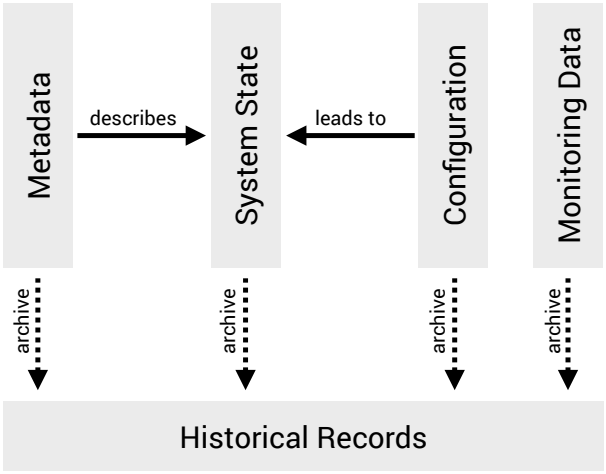

Figure 2.1: Overview of the information and parameters included in the network context.

We include *metadata* that describes the state representation as well as methods to interpret and convert the information if necessary. This metadata is required to interpret the state information correctly and to be able to re-use the stored information across different mechanisms as well as at a later point in time. Possible information can range from simple data types such as `Integer` or `String` to extended information on how to interpret the stored information correctly such as object definitions. Metadata might also include the function names (but not the function itself) to interpret, convert and transform the stored state information.

The context also holds the *configuration parameters* of the mechanisms since the state information usually depends on these parameters, and even minor changes to the configuration can lead to a complete other mode of operation and thus completely

different state. Static configurations are normally stored in files that are loaded on startup and thus are persistent between restarts, but cannot be altered during runtime. We not only capture the static parameters of the configuration, but also keep dynamic adaptations of the settings in the context. This allows us to provide a self contained view on the mechanism without referring to external sources such as configuration files.

We include *monitoring information* that is requested by external mechanisms into the network context. Monitoring information is not needed for the operation of a mechanism, but allows external services to police and intervene the operation if changes in either the configuration or the state are necessary. The inclusion of monitoring information enables us to generate individual views on the operational status depending on the requirements of the external mechanism. The monitoring information is usually closely coupled to the internal state and thus a tight integration allows us to gain additional insights into the current operational status.

Additionally, we include *historical records* for the information mentioned above. The history enables us to return to a known previous context of the mechanism that includes all relevant data for the operation. A rollback can be used to re-establish a certain setup where the internal state together with the configuration parameters provided a optimal operational environment.

Wherever the term *context* is used throughout this thesis, we specifically refer to the network context.

2.2 NETWORK ARCHITECTURE

The design and operation of large campus networks or Data Center Networks (DCNs) is a challenging task. While adjustments to the operational parameters are possible, fundamental decisions made during the design phase of such a network can often not be revised once the mechanisms are implemented and under operation. Risking a malfunction or even a complete downtime of the network is prohibitive as the mechanisms normally run under tight SLAs that guarantee customers a high availability in conjunction with a certain Quality of Service (QoS). Additionally, these networks usually run under a very high utilization and thus the traffic is not easily redirected over secondary links during the reconfiguration.

Deploying a completely redundant infrastructure—ideally using hardware from different vendors and operating the mechanisms by disjunct teams—to prevent outages is usually prohibitively expensive even for critical networks and only a few Tier 1 providers use a completely redundant setup.

However, with the wide spread deployment of SDNs and especially with the introduction of OpenFlow [49] as a standardized architecture, the task of dynamically reconfiguring network elements became less tenuous. The first large scale networks are now under full operation [33] and mechanism developers as well as operators gather first hand experience in designing and operating global SDN deployments.

Additionally, the introduction of SDN has led to new architectural concepts in network design and to completely new applications that are either run on top of or in direct interaction with the network. These applications were either not possible with traditionally operated networks or their deployment was a tedious and often expensive task.

Still, operating these networks requires to incorporate legacy mechanisms as the functionality is either not yet available in an SDN compatible mechanism or existing infrastructure has to be interconnected with the new deployment.

2.2.1 *A Knowledge Plane for the Internet*

Clark et al. [18] propose a new paradigm in network design and operations where a pervasive system is resonsible for 1. building and maintaining an abstract model of the overall network state, 2. orchestrating all network components to actually achieve the desired state, and 3. monitoring the network for problems that possibly result in a deviation from the desired state. This *Knowledge Plane* is construed of multiple loosly coupled mechanisms that cooperate in order to reach the global network state defined by the operators.

The system is designed such that parts—run on both end hosts and servers within the network—report back to a locally centralized instance that is responsible for monitoring and enforcing the local as well as the global goals. This local instance federates with other instances running in different parts of the network in order to gain a global view, and to infer the globally optimal configuration from all local requirements.

The authors argue that, by using Artificial Intelligence (AI) and cognitive systems, the Knowledge Plane enables the network to become self aware and independent of human operators. This independence allows the network to monitor and reconfigure itself in order to provide meaningful feedback to the end users and to optimize its operations within the boundaries of the operator defined ruleset.

The key principles of the architecture are the integration of data that is routed through the network to different points where the information is required, and the need to operate on incomplete or imperfect information. The latter is especially important as the information is collected from different administrative domains and from networks with different operational states. However, the architecture requires all connected mechanisms to support a single interchange format as it is missing support for transformation functions, and thus is not able to adapt itself to different representations of the same information.

While the work defines the requirements of such a Knowledge Plane and suggests possible building blocks, the system was never implemented nor evaluated. The authors thus fail to prove that their architecture is actually feasable for achieving the goal of a globally but decentralized management and coordination system that is able to orchestrate large scale networks such as the Internet.

2.2.2 *Routing Control Platform (RCP)*

The Routing Control Platform (RCP) [23] by Feamster et al. is a first step towards a software-defined routing as it introduces a separate control plane along with a central decision engine (a controller) for a network of BGP routers. The authors argue that the routing decisions should be decoupled from the actual forwarding and IP routers should be more like lookup-and-forward switches that forward packets as fast as possible. The RCP is connected to all routers in an Autonomous System (AS) and provides centralized routing decisions based on global knowledge.

The proposed separation allows for a network wide path selection based on expressive policies while keeping the changes to existing routers minimal. RCP enables a more flexible traffic engineering as the egress link for certain traffic can now be changed and the operators can configure a load balancing system to utilize all existing downstream paths, thus avoiding congestion on certain links that would normally be selected by the routing algorithm. Additionally, the proposed solution can enforce correctness contraints and invariants on the routing decision as it can ensure that each router along the forwarding path selects the optimal BGP route and thus forwarding loops are avoided.

As RCP uses BGP to connect to the routers, only configuration changes are necessary and the existing hard- and software can be left in place.

In a first step, only the intra-AS or Interior Border Gateway Protocol (iBGP) links are replaced with a connection to the RCP and the inter-AS or Exterior Border Gateway Protocol (eBGP) sessions are still handled by the individual routers. In this phase, the edge routers of a network need to apply the routing policies defined internally and distributed via the RCP to the traffic received via the edge links. The RCP acts similar to a route reflector but is able to provide individual results to each router.

In a second phase, the BGP sessions at the edge links are replaced by connections between the RCP instances of the different operators that run each AS. The individual instances continue to use eBGP to exchange routing information. However, due to the direct connection between RCP instances, the operators can apply global policy changes directly on the routing information gathered from other ASes instead of applying the policies locally at each ingress router.

On the one hand, the Routing Control Platform allows network operators to gain a global view on their network of routers while they can continue to use the existing hard- and software, such as routers, that provide hardware acceleration for the data plane. Also existing, and well known and understood protocols are used which allow operators to incrementally introduce RCP into the existing infrastructure.

On the other hand, RCP is still limited to the lowest common denominator in exchanging routing state and lacks the ability to seamlessly incorporate new and upcoming routing protocols.

2.2.3 *A Software-Defined Internet Exchange (SDX)*

The work on a Software-Defined Internet Exchange (SDX) [30] by Gupta et al. describes a novel architecture for introducing SDN into an Internet Exchange Point (IXP) environment where no single operator controls the network but networks of different Internet Service Providers (ISPs) with potentially different interests are interconnected.

Traditionally, IXPs such as DE-CIX [21] offer a neutral point for ISPs to exchange traffic without requiring a separate peering location. The IXP operates a switch fabric where all participating carriers connect their networks using edge routers. These routers are currently using BGP to exchange routing information and peer with other ISPs. The IXP usually also operates a BGP route reflector to reduce the number of active BGP sessions and to distribute the routing information among all participants.

The continuing increase in (video) traffic along with an increasing number of peering disputes between content providers and ISPs put the IXPs in the front line of providing technical solutions for a more sophisticated traffic management together with a flexible peering infrastructure. SDX thus offers a solution based on a flexible SDN infrastructure

to allow carriers as well as content providers to specify peering policies across the complete IXP infrastructure.

Simply deploying SDN-enabled hardware is not feasible in an IXP environment as the networking policies along with potential wide-area traffic-delivery applications are not run by a single operational instance but each ISP is defining its own policies and runs its own applications that support the specific traffic engineering goals of that carrier.

SDX provides an abstract network consisting of virtual switches to each ISP that accept the defined policies and commands issued by applications but do not directly enable these rules on the physical hardware. The SDX combines the policies issued by the different ISPs to a single coherent set of rules for the physical switches. This abstraction allows for isolation between the different ASes that are run by the participating ISPs without loosing the flexibility of a SDN.

As the SDN policies defined by the carriers must match the announced BGP path for traffic forwarding, the SDX allows carriers to define forwarding policies relative to the currently advertised BGP routes. The SDX therefore integrates a route reflector that allows each participant to forward traffic to all feasible routes for a prefix. The announced BGP routes are used as a default for forwarding traffic if no policies for the particular flow are defined and traffic is only routed along the BGP-advertised path such that a network that does not announce a prefix will never receive traffic for this destination. The tight integration of the route reflector with the SDX infrastructure also allows for policies based on BGP attributes and to distribute BGP routes based on the defined policies.

However, the SDX still relies on the features of an underlying BGP infrastructure and while it allows the definition of fine grained forwarding policies, these policies do not incorporate novel routing protocols nor can they be easily migrated to other platforms.

Additionally, the focus on a single use case along with the restriction of only supporting a single routing protocol—due to missing support for transformation functions—limits the advantages of the SDX architecture.

2.2.4 *Statesman*

Sun et al. [68] introduce a network-wide state management architecture (see Figure 2.2) that is tailored towards large DCNs. Due to the sheer size of these networks, problems arise that usually do not occur in smaller networks such as simultaneous failures of components or a sudden burst of traffic for a specific resource. DCNs thus usually run multiple management applications that are tailored towards a specific scenario such as component failure and traffic bursts. These applications often come with conflicting interests such as avoiding a specific path due to failure while routing traffic over this path to mitigate the burst.

Statesman provides a network-state management system that allows management applications for DCNs to export and import certain state to a central location in the network. The system functions as an invariant checker and conflict resolver. It focuses on the collection and migration of state from multiple network management applications. The goal is to manage the configuration state of the complete network and to allow for a coordinated network-wide state transition while keeping track of network invariants and offering several mechanisms for conflict resolution during state migration.

The underlying model abstracts the network state into three views: 1. the observed state, 2. the target state and 3. the proposed state. The observed state represents the

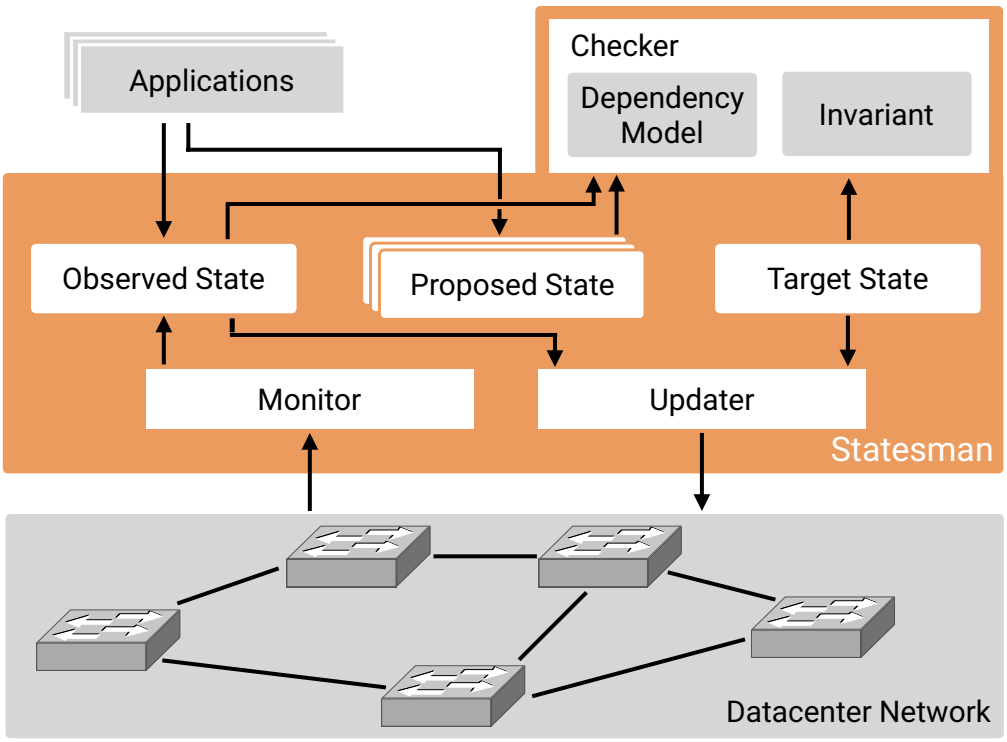

Figure 2.2: Statesman architecture (adapted from [68])

current state of the network while the target state is the desired configuration of the network which Statesman is responsible to reach. The poposed state is introduced to capture the state desired by the applications and each application writes its own proposed state to the system. Statesman then detects possible conflicts between the proposed states and validates them against a set of network-wide invariants which capture the basic operational conditions such as required backup mechanisms or maximum link load. The states are accepted and merged into the target state if the proposed states are not conflicting with each other and are compliant with the invariants. Those changes are then executed on the network.

Statesman focuses on the network-wide configuration state of management applications, but is not designed to handle the state of running mechanisms. It also requires applications to adapt their internal state representation to the format defined by Statesman and thus potentially requires major changes to the underlying data structures.

2.3 CROSS LAYER SYSTEMS

Traditionally, network stacks follow a strictly layered design where only mechanisms on adjacent layers are able to communicate with each other and the exchange of information is limited to the absolute minimum. Typical implementations of this approach are the OSI model [77] and the TCP/IP model [13, 14] which itself is based on the DoD model [16].

While limiting the communication to neighboring layers allows for a clean definition of the communication model as well as the Application Programming Interfaces (APIs), it also limits the possibilities for adaptation of mechanisms running in each layer, and some even consider the strict layering of network mechanisms harmful [15].

Cross layer systems aim to remove the strict separation of mechanisms in the different layers of the network stack, and allow for information exchange between layers not directly adjacent to each other. This is either done by introducing additional communication paths that directly connect mechanisms in non-adjacent layers, by providing a separate module that spans multiple layers or by completely removing the layered approach and introducing a new paradigm for mechanism abstraction [66].

While we acknowledge the existence of other approaches in the field of cross layer architectures, we focus on systems that introduce a separate element for collecting and disseminating information from mechanisms across the network stack as these approaches are closely related to the goals of the thesis defined in Section 1.2, and specifically the contribution described in Section 1.3.1.

2.3.1 *MobileMan*

MobileMan [19] by Conti et al. is a cross layer architecture for WMNs that allows the cooperative sharing of status information across all layers of the network stack without sacrificing the separation of mechanisms into layers during the design phase. This allows mechanism developers to implement mechanisms that run independent of each other while offering the possibility to benefit from information gathered on other layers.

The main goal of the enhanced architecture is to increase the local communication and information exchange among mechanisms and thus reducing the need for remote communication which subsequently leads to an optimized use of bandwidth and energy.

The authors introduce an additional module named *Network Status* that spans across all layers of the network stack as depicted in Figure 2.3. The module is responsible for handling the state information of mechanisms running on each layer as well as providing methods for direct communication between those mechanisms. The Network Status holds information that are usually duplicated across multiple layers, thus enabling mechanisms to re-use the data gathered on other layers, and allows for a de-duplication of information.

MobileMan combines the advantages of a layered design with the advantages of a cross layer architecture, and allows network designers to combine existing mechanisms in some layers that are not yet cross layer enabled with mechanisms in other layers that are already built with information sharing in mind. This allows for full compatibility with existing standards as the core functionality of each layer is not modified, and the benefits of a modular architecture are maintained.

However, existing mechanisms need to be heavily modified to benefit from the MobileMan architecture as the internal state representation needs to be adapted to the interfaces and the data model provided by the Network Status element. Additionally, when developing new mechanisms, the designers need to strictly adhere to the specifications imposed by the architecture to be able to share state information, and are thus possibly forced to use less efficient internal data structures to be able to share information using MobileMan.

Additionally, the architecture is limited to the network stack of a single host and is not able to extend the sharing of information across multiple mechanisms that are distributed in a network segment. While this allows for a fast exchange of information due to the local nature of the data structures, the limitation to a single node highly restricts the possible gains of sharing state information as a regional—within a network segment—or even global optimization is not possible.

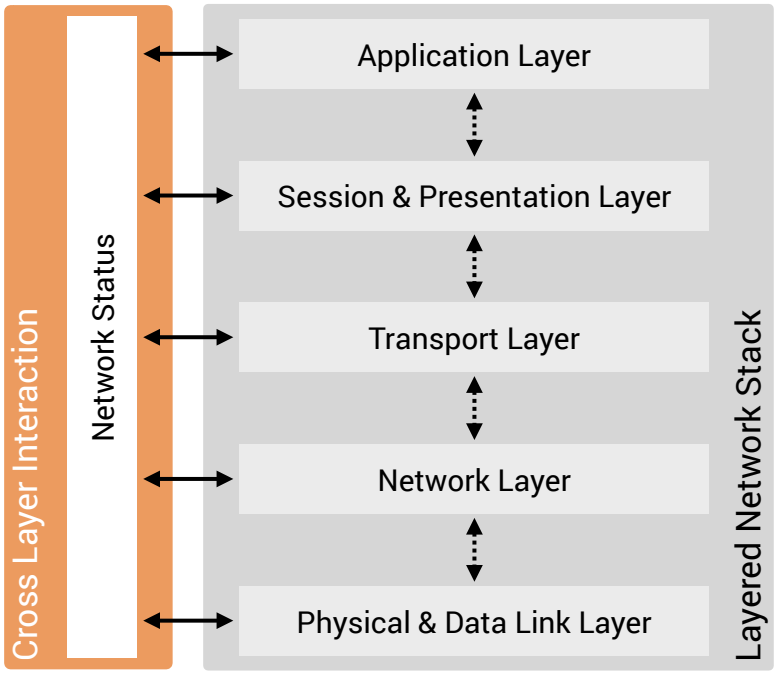

Figure 2.3: MobileMan architecture (adapted from [19])

2.3.2 *CrossTalk*

Winter et al. propose CrossTalk [76], an architecture that not only focuses on the dissemination of locally gathered information. It also includes globally collected data to optimize the behavior of mechanisms. The goal is to not only provide local improvements of the running mechanisms within a single network stack but to increase the overall performance of the complete network by including a global view into the local optimizations.

CrossTalk includes two additional elements that span all layers of the networking stack as shown in Figure 2.4: 1. The *Local View* is responsible for gathering and distributing local information, and 2. the *Global View* is used to collect information provided by other nodes in the network and to distribute this global data to local mechanisms. The global information is piggybacked on existing packets to reduce the number of packets that need to be transmitted and to minimize the overhead for the global data exchange.

Additionally, the Local View does not only include information provided by the mechanisms running in the local network stack but also by other system components such as the OS, hardware components etc. This tight integration with the nodes' components allows for even better optimizations on the local level, and—together with the global dissemination of information—enables the sharing of these data within the complete network to achieve a global optimization of the connected mechanisms.

The architecture requires the connected mechanisms to fully adapt the data structures and semantics provided by CrossTalk in order to benefit from the shared information. This forces developers to inherit the structures provided by the architecture instead of selecting the optimal specifications based on the requirements of each mechanism.

Additionally, CrossTalk duplicates the global information across all nodes of the network which in turn can lead to inconsistencies in the stored information and the

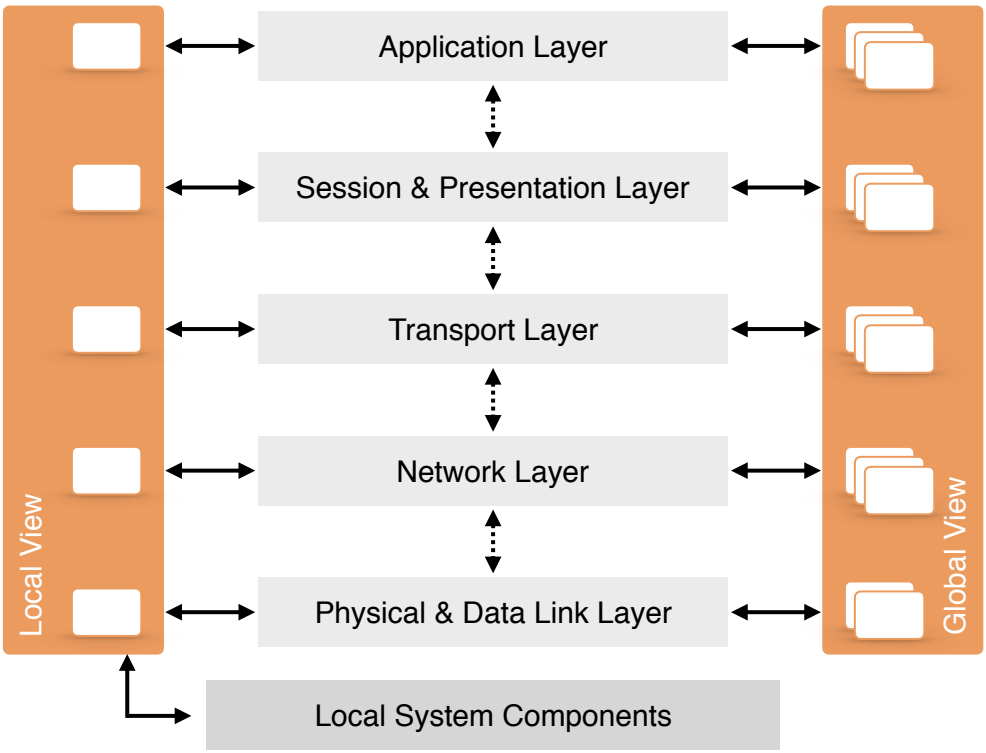

Figure 2.4: CrossTalk architecture (adapted from [76])

decisions based thereon. This is especially critical in networks with high dynamics where the data dissemination using the piggybacking mechanisms takes longer than the validity of the information.

2.3.3 *CLiSuite*

Instead of providing direct cross layer interaction between mechanisms, CLiSuite [44] by Lindeberg et al. introduces a middleware that abstracts from the concrete implementation dependent representation of state information. The framework instead offers a generic approach to access and re-use state provided by other mechanisms by mapping semantic equivalent information to a common *network state*. It thus provides mechanism independent state information that is dynamically mapped to the mechanism specific data.

CLiSuite consists of four major components that collect data from the mechanisms on all layers of the network stack, transfer the information into an abstract state representation and disseminate the abstract information to the mechanisms again. The overall architecture is depicted in Figure 2.5.

Each mechanism needs to implement the *CLiWrapper* interface that is providing a publish-subscribe interface to transfer the state information within the mechanism to the CLiMonitor. This interface abstracts from the internal representation of the mechanism state and provides a common interface to the other components that is not dependent on a specific implementation.

The *CLiMonitor* functions as an event bus between the mechanisms running within the network stack and the storage component. It receives atomic events from the connected

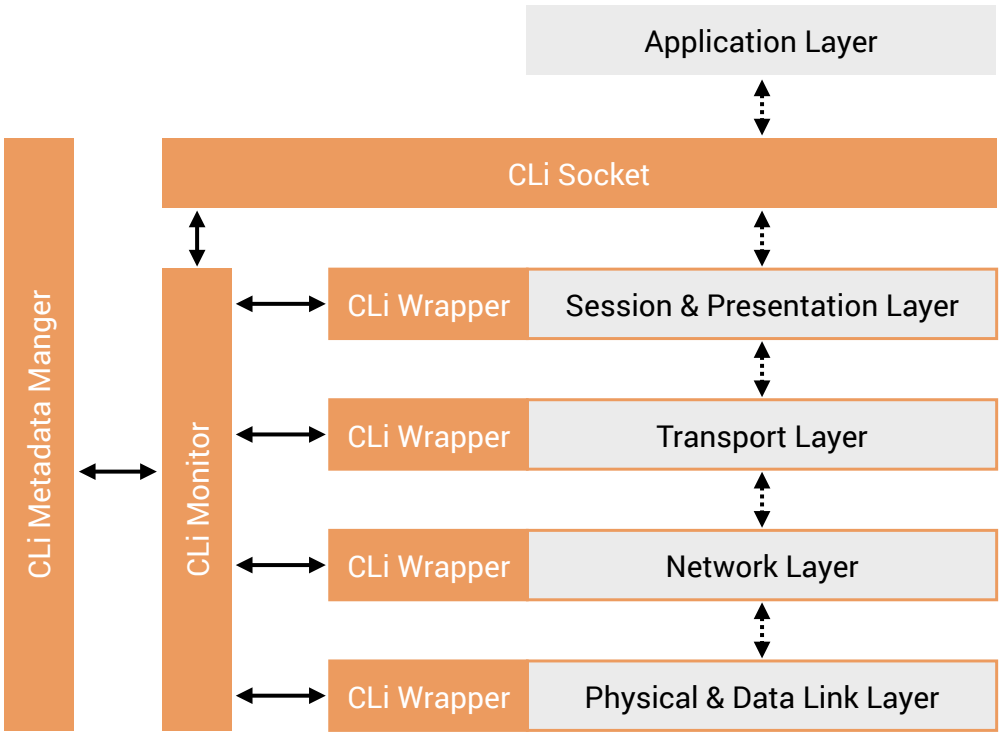

Figure 2.5: CLiSuite architecture (adapted from [44])

mechanisms on state change, and is able to forward already gathered information to the requesting layer.

The storage for network state is represented by the *CLiMetadataManager*. It holds the abstract state information, and is responsible for converting between the abstract representation used to by CLiSuite and the concrete representation required by each mechanism.

The *CLiSocket* provides applications running on top of the stack with an interface that supports the primitives of BSD sockets and acts as an compatibility layer between the—possibly unmodified—application and the cross layer-enabled network stack. Additionally, the socket offers applications an API to interact with the CLiMonitor and access the CLiMetadataManager.

CLiSuite also provides filters to limit the amount of information that is stored as Network State, and reduce the load on the CLiMonitor. However, currently only simple filters such as predicate and aggregate operators are available which limits the possibilities of selecting the required state information.

Additionally, CLiSuite only allows for sharing information between mechanisms that are running on the same network node and even within the same network stack. The architecture does not provide facilities to share state information across multiple nodes in a network, and thus cannot provide regional or even global optimizations. The mechanisms are also not able to incorporate information that are collected outside the actual data transfer such as energy levels, environmental conditions or OS status.

2.4 ROUTER MIGRATION

Routers are at the heart of the Internet. They are the devices that allow the interconnection of different ASes and thus enable the network of networks in the first place. The BGP protocol [59] is used to exchange information with both routers in the same as well as in adjacent networks. It was designed in the early stages of the Internet, and today's requirements for high availability and seamless migration were not in focus of the protocol designers.

Thus, the migration and replacement of BGP routers is a tedious task that requires long and accurate planning. The sessions between routers run indefinitely and only allow for very short interruptions until the routing entries are invalidated [72].

However, a replacement for BGP is currently not in sight and even the design of modern networks is limited to this protocol as the common denominator for exchanging routing information with other networks.

The need for dynamic adjustments of the routing infrastructure has lead to various works that try to tackle this challenge. In the following, we discuss the most important ones.

2.4.1 *Virtual Routers on the Move (VROOM)*

Wang et al. [73] propose Virtual Routers on the Move (VROOM), a system that decouples the control plane of routers from the data plane and introduces a *dataplane hypervisor* that allows the router to freely move from one physical system to another without interrupting neither the data nor the control traffic.

VROOM partitions a physical router to create virtual router instances that each run their own control and data plane and are thus able to forward traffic independently of each other. Each control plane can run in a container or virtual environment, executing its own applications, routing protocol implementations and configurations, while the independent data planes all reside on the substrate, but each holding a separate Forwarding Information Base (FIB) and operating on separate interfaces. This separation enables VROOM to migrate the control and data plane independently of each other and thus support a seamless move of (virtual) routers between hardware instances.

The seamless migration is assisted by a *remote control plane* that forwards protocol updates received by the migrate-from router to the moved control plane such that the router remains reachable by its adjacent protocol peers and can receive update notifications. The mechanism also allows an already migrated control plane to send updates back to the data plane of the migrate-from router and thus keep the forwarding information current even during migration.

Additionally, VROOM introduces the concept of *double data planes* where a single control plane can orchestrate two data planes, one residing on the migrate-from and one residing on the migrate-to router. This aids the seamless migration process as it allows for asynchronous link migration where some traffic still flows trough the old router while other traffic is already handled by the new instance.

However, VROOM is designed to migrate routing state between router instances of the exact same type. The authors leverage this limitation to speed up the migration time by providing a basic Virtual Machine (VM) image on each physical instance and only

copying the configuration and runtime state in the control as well as the data plane between routers.

2.4.2 *Router Grafting*

Router Grafting [37] by Keller et al. is a system that allows to seamlessly remove partial state of a BGP router from one mechanism and merge them into another. The system focuses on typical use cases in large provider networks that connect multiple customers and where SLAs often prevent operators from reconfiguring parts of the networks as this reconfiguration would lead to customer downtime and thus violate the SLA.

Router Grafting performs the following steps to achieve a seamless migration of single BGP sessions: 1. export the protocol configuration on the migrate-from router 2. migrate the underlying link, 3. migrate the underlying Transmission Control Protocol (TCP) connection, 4. import the protocol configuration on the migrate-to router, and 5. exchange any routing state that was changed during the migration process. While the steps two and three are covered by existing solutions, namely SockMi [11] and programmable networks such as SDNs, the first and the last steps require modifications of the BGP implementation.

The authors modified the BGP state machine and introduced an additional *inactive* state (see Figure 2.6) to allow the router to import the necessary information without trying to comunicate with the remote end-point. Once the migration of the configuration and link state is completed, the router transitions into *established* state and continues to import the existing routing information extracted from the Routing Information Base (RIB).

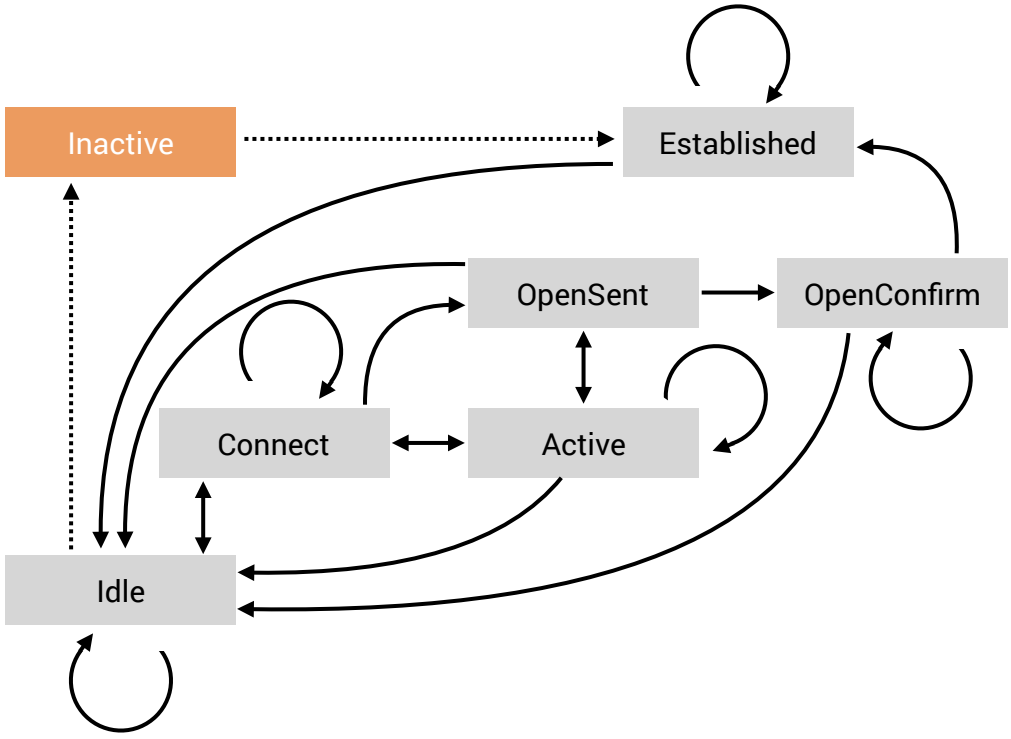

Figure 2.6: Extended BGP state machine used by Router Grafting. The additional inactive state is highlighted.

Additionally, the migration requires the BGP decision process to be re-run, and routing changes to be disseminated to neighboring routers, once the state is transfered. This step is necessary as the migrate-to router might have another best route for the destination prefixes. To avoid a complete reconfiguration of the neighboring routers, the migrate-from router replays the messages received from the neighbor such that the migrate-to router is able to run the decision process and notify all other connected mechanisms about the necessary updates.

The system currently only handles the RIB as the most important source of state information available at the router. All other state such as timers and statistics are reinitialized on the migrate-to router during the migration and the information gathered so far is lost.

Router Grafting specifically targets the migration of routing state between mechanisms from different vendors where interoperability is usually not given. Therefore, it introduces the possibility to adapt the format of the RIB during the migration. However, this adaptation is performed offline and thus takes an additional step in the migration process.

2.4.3 *Seamless BGP Reconfigurations*

Vissicchio et al. [71] propose a scheme for seamlessly reconfiguring BGP routers named *Ships-In-The-Night (SITN)*. The system is based on the Virtual Routing and Forwarding (VRF) feature [62] that is available in most commercially available routers, and thus requires only minimal changes to the existing infrastructure. VRF provides multiple namespaces within a single router, and allows operators to run multiple independent RIBs on a single system.

SITN maintains multiple RIBs per device, but only one routing table is active at any time. This allows for updating the inactive routing tables on all devices of a given AS without any effects on the actual forwarding as well as the eBGP sessions maintained with external mechanisms.

The architecture requires three main components to achieve a seamless reconfiguration: 1. a mechanism to propagate all external routes to multiple namespaces, 2. an interface that propagates iBGP updates from the currently active namespace to the eBGP peers, and 3. a tagging mechanism that is able to label packets with VRF information. While the first component is available in commercial routers, the latter two are implemented using a BGP proxy that is located in front of the actual router as shown in Figure 2.7.

The proxy terminates the eBGP sessions from external peers and maintains an iBGP session per VRF domain to the router. The proxy provides a BGP multiplexer [70] but extends the architecture to support the concept of active namespaces—which allows to switch between different RIBs—as well as selective update propagation.

While SITN provides seamless BGP reconfiguration within a network, it does not directly support the migration of the complete RIB between router instances. Additionally, SITN is a special purpose solution that only focuses on BGP, and is not generally applicable to either other routing protocols or even mechanisms from a completely different domain.

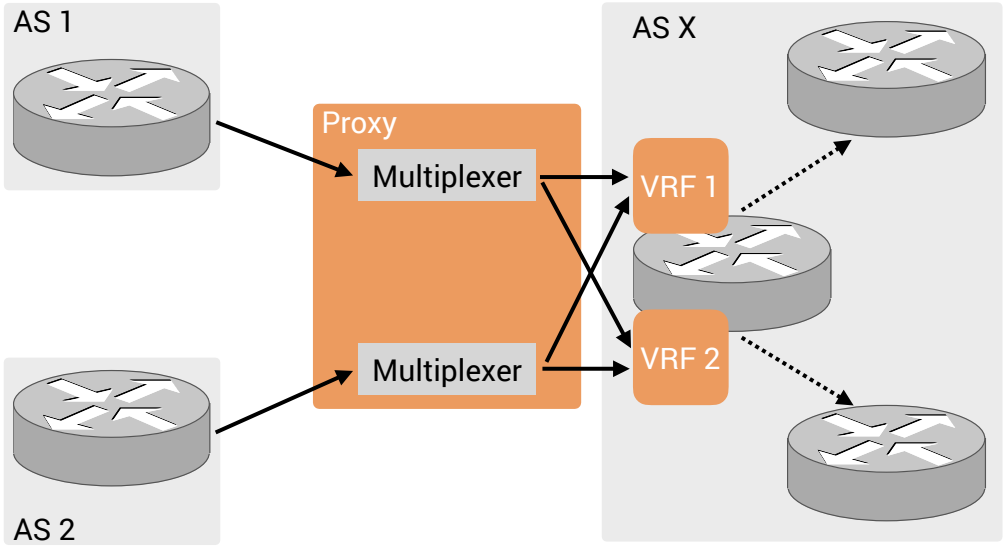

Figure 2.7: Ships-In-The-Night architecture (adapted from [71])

2.5 NETWORK FUNCTION VIRTUALIZATION

Network Functions pervade networks of various sizes from small company networks running a single NF to large DCNs where NFs already make up between 30 % and 50 % of all networked devices [64, 65]. NFs range from simple mechanisms performing Network Address Translation (NAT) over packet filters and firewalls to advanced functions such as Intrusion Detection Systems (IDSes) and Deep Packet Inspection (DPI) that monitor and manipulate traffic on all layers.

Traditional NFs are vendor specific devices that function as a black box to the operator and can only be configured and customized to a pre-defined level. These devices are placed statically in a network path and features such as scalability and High Availability (HA) come at an additional operational and monetary cost, if they are available at all.

However, research efforts in this area have lead to service oriented [6, 26], composable [63], extensible [7] and scalable [28, 57] NF implementations. The introduction and combination of technics well known from other areas of network design and operations, namely SDN and hardware virtualization running VMs, fostered the development and deployment of NFV.

NFV uses virtualization on commodity servers to enable dynamic scaling of NFs and SDN principles to compose new functions by chaining existing NFs. This approach in deploying NFs has lead to the development of Service Oriented Architectures (SOAs) in the area of network design and operations where each function only executes a small but specific task and multiple functions are dynamically combined to achieve an overall goal.

Currently, there exist two types of systems that are either optimized to provide HA or to enable dynamic scaling of virtualized NFs. In the following, we discuss both categories with a focus on the possibilities and methods for sharing state information.

2.5.1 *Remus*

The goal of Remus [20]—proposed by Cully et al.—is to provide transparent recovery from fail-stop failures of a single physical host, thus enabling HA on commodity hardware for existing, unmodified software. It does so by snapshotting VMs on a regular basis and transferring this snapshot to a backup mechanism holding a secondary copy that is spun up upon failure of the primary instance.

The system is based on the Xen [9] hypervisor, utilizing, optimizing, and extending the existing functionality to seamlessly migrate VMs between hosts. It runs paired servers in an active-passive setup where one mechanism takes the role of actively executing the replicated VM while the passive mechanism only manages and stores the replicated disk and memory content until the active instance fails. Upon failure, the passive instance immediately takes on the active role without a boot process as all necessary information is readily available. This requires a high snapshot frequency to ensure that the state information on the passive instance is up to date, usually leading to a high operational delay as the memory access is basically limited to the speed of the synchronisation network.

Therefore, the authors explicitly do not to attempt to achieve a deterministic behavior of the replicas. Instead, they accept that the output can be different, even when the same input is replayed and use *speculative* execution [51] ahead of the synchronisation points. Buffering the output of the active mechanism enables the operator to define a tradeoff between consistency and output delay depending on the requirements of the network and the application running inside the VM. The buffered output also enables Remus to use asynchronous state replication that allows the active instance to continue operating even if the snapshot is not yet committed on the passive node.

The output buffers introduce additional burstiness into the network traffic as the output is now released at intervals instead of as a continuous stream of data.

There is no need to specifically design software to use Remus but it supports any OS and application without modification to run inside the replicated VM. However, directly copying memory between the active and passive instance limits Remus to running exactly the same implementation not only of the application providing the functionality but of the complete VM including the OS and any auxiliary software.

2.5.2 *Tardigrade*

Tardigrade [46] by Lorch et al. is a system for deploying existing, unmodified binary applications as fault-tolerant services using Lightweight Virtual Machines (LVMs). Similar to Remus, described in Section 2.5.1, it asumes a fail-stop model where host machines only fail by completely stopping operation but not by acting arbitrarily.

The LVMs employed by Tardigrade only use a thin layer of abstraction between the host OS and the virtualized application stack instead of replicating a complete VM including the OS running an independent kernel, managing different memory modes and executing background services. This reduces overhead for both network latency and replication bandwidth as the memory footprint of LVMs is much lower, resulting in faster snapshot creation and smaller snapshots to be transfered between the replicas.

Additionally, the system is not limited to a single pair of servers but supports multiple replicas of the same LVM that all act as a passive mechanism to the one active instance

processing packets. On failure, one of the passive mechanisms takes the active role and the other replicas continue to receive and process state updates. This enables Tardigrade to not only handle the failure of a single host but continues to offer HA even if multiple servers fail.

The architecture of Tardigrade is shown in Figure 2.8. It is based on three main components: The *orchestrator* is responsible for coordinating both failure detection and instance management as well as snapshot coordination. It uses an unreliable failure detector that receives periodical heartbeat messages from each instance and decides, based on these updates, if a fail-over should be initialized (failure of the currently active instance) or an instance should be removed from the set of backups (failure of a passive instance). The orchestrator also manages the creation and distribution of snapshots among the instances. It distinguishes between *full* and *incremental* snapshots. While a full snapshot can always be applied to an instance, an incremental snapshot is only applicable if the last full snapshot and all subsequent incremental snapshots have been applied. The active instance only proceeds operation—i.e., releases buffered network packets—if the generated snapshot is applicable to all backups.

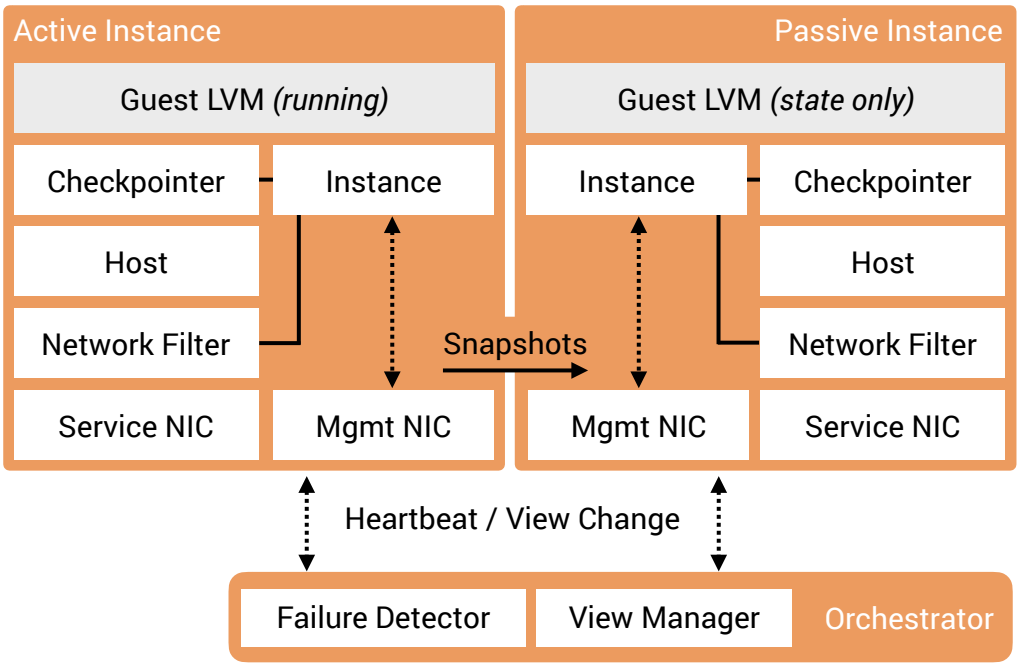

Figure 2.8: Overview of the Tardigrade architecture (adapted from [46])

The *checkpointer* is responsible for creating consistent snapshots across all system objects such as files, threads or memory locations. To track memory changes, it identifies the pages altered between two snapshots to efficiently generate incremental snapshots by intercepting the memory write operation and setting a dirty bit in its own metadata. For file-change tracking, the checkpointer captures ongoing write operations that are replayed on the backup without the need to track the actual changes in a file.

The *network filter* is the third major component within Tardigrade. It is responsible for handling the buffering and release of network packets on the active instance and ensures that only the active instance is visible towards the network using the shared address. Due to the socket based implementation of the network interface in Bascule [10]—the LVM

framework used by Tardigrade—the TCP connections are currently broken on failover as the TCP state is not replicated between instances.

While Tardigrade uses LVMs and thus the footprint of the replicated memory is small, the authors recommend to use the system only for services that have a low memory-dirtying to bandwidth ratio. Otherwise the snapshotting of a LVM and the replication of a snapshot takes too much time and the network latency significantly increases due to buffering the output until the snapshot is committed.

2.5.3 *Split/Merge*

Split/Merge [57] by Rajagopalan et al. provides an abstraction that uses state replication on scale-out and state merge on scale-in. It uses a state-centric, system-level design for seamlessly scaling NFs. The system makes use of the specific state properties of NFs that each instance is only responsible for a certain number of flows and thus only needs to hold state information for that specific set while global state shared between all flows usually is neither large nor critical for the operation of the NF.

The state information is classified into *internal state* which is only required by a instance to run but has no effects on the migration (e.g., cache entries, temporary information), and *external state* that needs to be migrated in order for the NF group to function properly. The external state must be consistent across all NFs that access the information to ensure consistency and loss-free operation. The external state is further classified into *partitioned* and *coherent* state. While the first class is exclusively accessed by a single instance (e.g., flow specific information), the latter is shared globally between all instances of a NF.

Split/Merge provides a library that externalizes state management and acts as an additional layer in accessing the stored information. It provides functions to handle per-flow (external partitioned) as well as shared (external coherent) state on a transaction basis where state information must be locked before access and must be unlocked after operations are finished. The library is also responsible for copying partitioned sub-state to another replica when notified by the orchestrator.

The *orchestrator* is a central instance that coordinates the complete operation of Split/Merge similar to the controller in a SDN. It is supported by an agent that runs on the virtualization host and manages the dynamic creation and destruction of VMs.

The system uses an underlying SDN to ensure that packets are always forwarded to the VM responsible for the flow the packet belongs to. The orchestrator has access to the SDN and can implement rules that ensure the forwarding policy. Split/Merge relies on certain configurations of the underlying network and the VMs. For example, all VMs must share common MAC and IP addresses.

Split/Merge is designed to operate loss-free but explicitly allows for reordering of packets even during normal operation. It is up to the NF and the subsequent mechanisms to cope with the reordered data stream. To ensure loss-free operations, Split/Merge halts all flows and buffers the packets at the orchestrator while migrating flow state. The buffering can lead to a significant delay of packets during migration and—depending on the current network load—might cause a large memory overhead on the orchestrator.

NFs must be explicitly designed to support the Split/Merge abstraction and must use a five tuple to identify the flow state. Split/Merge is limited to sharing state information between instances of the exact same implementation and the requirement of strong

consistency across coherent state information might cause a large overhead due to state synchronisation across multiple instances.

2.5.4 *Pico Replication*

Rajagopalan et al. also propose Pico Replication [56], a HA framework that is specifically tailored towards NFs. It operates on a per-flow basis and is able to create snapshots of the individual flow state instead of snapshotting the complete mechanism as done by other systems. This enables Pico Replication to achieve a higher snapshot frequency while having a lower overhead than VM centric systems.

A high snapshot frequency without introducing traffic burstiness is only possible by handling flow state individually as snapshotting requires packets to be buffered until the snapshot is committed. This prevents inconsistent state between NFs.

The framework utilizes concepts introduced by Split/Merge (see Section 2.5.3) and extends the existing mechanism with additional modules running on each host to support HA functionality. An overview of the architecture is given in Figure 2.9.

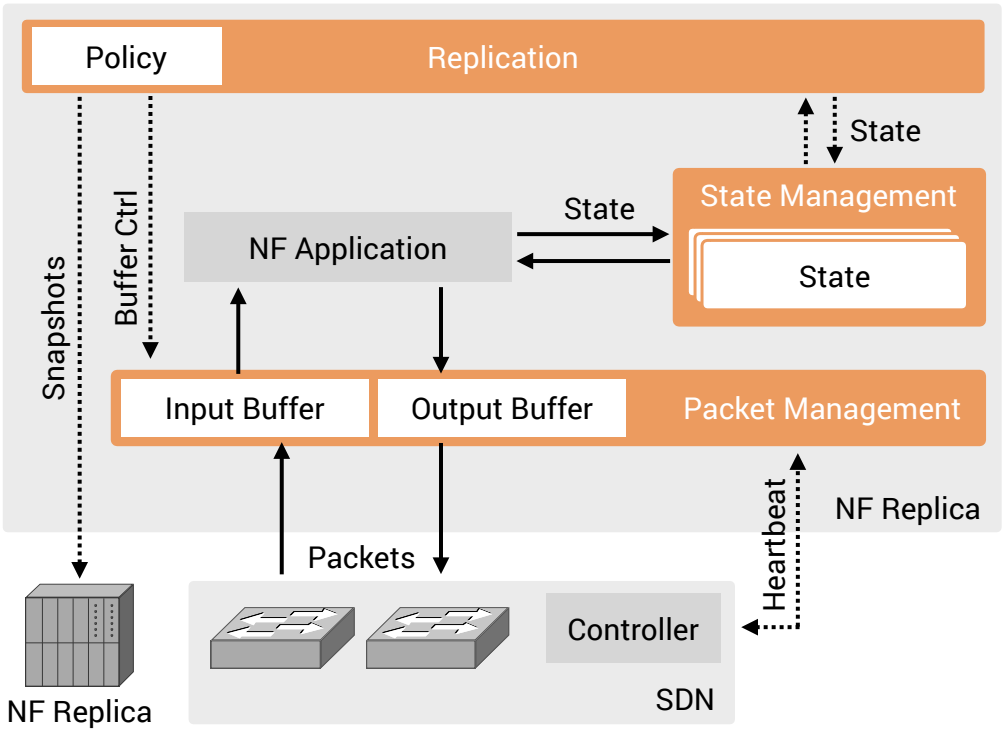

Figure 2.9: High level architecture of Pico Replication (adapted from [56])

In particular, Pico Replication introduces a *State Management Module (SMM)* that is responsible for identifying relevant information within a NF implementation and controls access to the flow state during operation. Pico Replication only targets *partitioned* state, that is state critical for the correct operation of a NF and that cannot be recreated during or after failover. The authors also explicitly exclude *coherent* state (e.g., static configuration, non-critical counters globally shared between NF instances) as it is assumed that this state information is already replicated by other mechanisms anyway.

The SMM contains both active as well as passive flow state. The *active* state is associated to a flow currently processed by the NF running on the host while the *passive* state contains information replicated to the system but belonging to a flow that is processed on another host (backup state). The passive state is not released to the NF until a failover is signalled and the instance is instructed to take over the particular flow.

Additionally, the SMM maintains *transactions* for each active flow state, thus keeping track of packet processing of the NF and therefore accessing or altering the state information. This tracking is necessary to define if it is save to replicate flow state or if the state is currently modified by a processed packet.

The *Packet Management Module (PMM)* is responsible for buffering packets during a snapshot operation. It buffers incoming packets during snapshot creation, thus suspending any operation on the flow state until the transaction is completed. This prevents state changes during operation as the state of NFs is solely dependent on received packets. Output packets are also buffered during snapshotting until the state is committed to the passive node. This is necessary to prevent state update in subsequent mechanisms that might become inconsistent if the NF instance fails during the snapshot.

The *Replication Module (RM)* orchestrates the operation of Pico Replication by executing a per-flow replication policy defined by the NF operator. It instructs the PMM to halt a flow, signals the SMM to create and transfer a snapshot of the state of the halted flow, and after receiving confirmation that the snapshot was transfered successfully, commands the PMM to release the output buffer.

The framework also utilizes the features of an underlying SDN to detect NF failure and redirect the assigned flows to a backup instance. The former is done by listening to *port down* messages of the switch that connects the NF instance to the network and the latter is done by rewriting forwarding rules in the SDN controller.

Pico Replication is specifically designed for HA requirements of NFs. It is able to continuously replicate state information between instances of the same implementation but lacks features to failover between different implementations. The dynamic scaling of NFs is not in focus of the framework and the authors explicitly refer to other solutions like Split/Merge that can be operated in conjunction with Pico Replication. Additionally, Pico Replication explicitly accepts temporary packet loss during failover as long as the end to end connectivity in upper layer mechanisms is retained.

2.5.5 *OpenNF*

OpenNF [28] by Gember-Jacobson et al. provides a framework that allows for "efficient, coordinated control of both internal NF state and network forwarding state". The framework enables operators to access and migrate the state of one NF instance to another instance while coordinating the packet forwarding in an SDN-enabled network to the currently active NF. This tight coordination enables OpenNF to avoid loss or reordering of packets, and thus provides guarantees on both latency and memory overhead during the move. The framework is able to satisfy tight SLAs on NF performance and availability while minimizing operating costs.

The setup uses a design similar to SDN as shown in Figure 2.10. A central controller coordinates both the state migration and the coordination of flows. On top of this controller, several applications provide the actual logic that initiates the migration and manages the state information during the operation. The application is connected to the

controller using a northbound API and can issue a *move* command to migrate the state of flows from one NF instance to another. Typical controller applications are performance or failure monitors that—in the first case—surveil the CPU and memory consumption of a NFV host and initiate a flow and state migration in case of high load on the system, or—in the second case—provide a hot standy-by mechanism that takes over the operation if the primary instance fails.

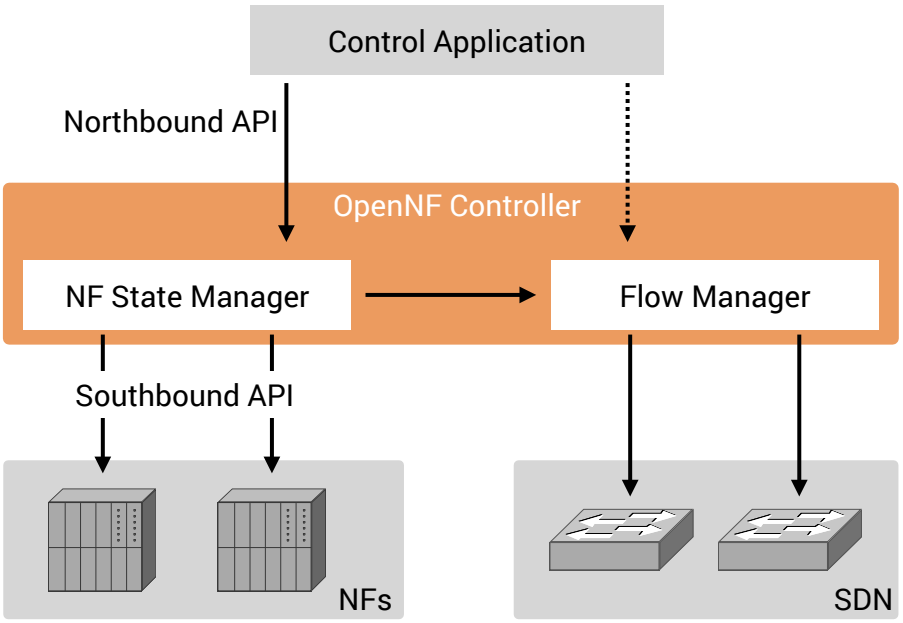

Figure 2.10: OpenNF architecture (adapted from [28])

The application can specify several modes of operation such as *parallelization* and *early release* of flow state information that allow for faster migration of NFs and thus keep the operational overhead low. This is especially important for virtualized NFs that run on highly utilized hosts and where the NF performance is critical for the operation of the network.

Depending on the requirements of the migrated NF and the specification given by the application, multiple levels of guarantees such as *no guarantees*, *loss-free*, *order preserving* and *consistency*, or a combination thereof ensure that the reconfiguration of the data network during the migration does not affect the operation of the NF. An IDS for example might rely on receiving packets in the correct sequence in order to accurately monitor the network for suspicious flows on the one hand but not raise false alerts due to packet reordering on the other hand.

The southbound API connects the controller to the actual NF and provides an interface to extract and insert state information. The controller is able to specify exactly which state to export and import and the API provides several functions to handle state information for single as well as multiple flows. Additionally, *filters* allow the controller to specify which flow state information should be extracted from the NF.

Depending on the mode of operation, packets are buffered at the controller which might introduce a high load on the central system and hinders scalability.

OpenNF focuses on the migration of state between the exact same implementation of a NF. It transfers the raw state information between instances and is thus limited to NFs that are able to directly process the transfered data. It is not possible to migrate

certain flow state to another implementation of the same NF that is better suited for the current traffic type such as specialized IDSes that are optimized to detect certain attacks. Furthermore, OpenNF is—due to missing support for transformation functions—not able to exchange state between different NFs such as sharing information about malicious flows between IDSes and firewalls.

2.5.6 *Distributed State Transfer (DiST)*

Kothandaraman et al. [40] propose a system that extends OpenNF—presented in Section 2.5.5—to support Distributed State Transfer (DiST). The enhanced framework removes the controller from the critical path during migration by using the data network to exchange state information directly between NFs. Removing the control network from the critical path—using it only for signalling—reduces the migration time and prevents controller as well as network overload, thus improving the scalability of the complete setup.

DiST extends the OpenNF framework by introducing additional commands such that the controller does not have to extract flow information from the migrate-from NF and then insert it to the migrate-to NF. The system is able to initiate a *move* that commands the migrate-from NF to directly send its current state to the migrate-to NF. It does so by extending the per-flow messages with the Internet Protocol (IP) address of the destination VM in the data network.

To allow for loss-free and order preserving operation, DiST also introduces packet forwarding between the migrate-from and the migrate-to NF. It uses OpenNF events to notify the migrate-from NF which packets to forward to the migrate-to instance. A packet buffer at the migrate-to NF holds the received packets until all relevant state is transfered. This can either be initiated on a per-flow basis where only packets for the currently migrating flow are redirected or on a global basis where all packets are redirected and the buffer is only released after all state is transfered.

While buffering is sufficient for loss-free operations, DiST introduces an additional mechanism to provide order preservation of packets. The system uses *InBand Control (IBC) Packets* to synchronize the two packet streams, one from the migrate-from NF and one directly from the switch. Once the migration starts, the switch is configured to duplicate the data stream and send one copy to each of the involved NFs. The switch then generates an IBC packet and inserts it into both data streams. Depending on which IBC packet is received first by the migrate-to NF, the packets are either buffered or directly processed until the second IBC packet arrives.

While DiST reduces the overhead imposed by OpenNF during migration, it does not remove the other limitations, namely sharing information only between NFs of the exact same implementation.

2.5.7 *P2P OpenNF*

Gember-Jacobson and Akella [27] introduce two mechanisms to improve the performance of a loss-free migration of NF state.

On the one hand, they remove the controller from the critical path by enabling NFs to directly transfer state information as well as packets between instances. On the other hand, they propose the reprocessing of packets to update the already transfered

state information while migration is still in progress. The optimizations are applied to OpenNF (see Section 2.5.5) but can be transfered to other state migration systems such as Split/Merge.

To support a peer-to-peer (P2P) state and packet transfer between NF instances, the authors extended the OpenNF southbound API to support two new functions, namely *accept* and *transfer*. The former instructs the migrate-to instance to accept incoming state transfers while the latter initiates the export of state information and instructs the migrate-from instance to duplicate and forward all packets to the migrate-to instance for reprocessing. The received data packets are either buffered on the migrate-to instance until the state transfer is completed or directly are forwarded to the NF for (re)processing. The extension uses virtual interfaces in conjunction with bridges to inject the P2P transfered packets into the packet stream received from the distributing switch.

The P2P forwarding of packets between NF instances significantly reduces the load on the controller and thus reduces the possibility of loss of processing performance during a migration due to controller overload.

The authors introduce the reprocessing of packets as a method to avoid packet loss during a migration due to overflowing buffers. Therefore, the migrate-from instance continues to process the incoming packets belonging to a flow even if the corresponding flow state is currently migrated while also forwarding packets to the migrate-to instance where they are buffered. The migrate-from instance only forwards packets that trigger a state update, reducing the number of buffered packets on the migrate-to instance, and consequently reducing buffer demands. The forwarded packets are then processed twice: once at the migrate-from instance and once at the migrate-to instance. While the migrate-from instance continues to process the packets and produces output as normal, the migrate-to instance only processes the packets to update the already migrated state information. It is thus necessary to suppress any output (both log entries and output packets) from that instance in order to avoid duplicate results. This is done by replacing the standard Input/Output (I/O) as well as the socket and the Packet Capture (PCAP) functions by stubs that either surpress the output on reprocessing or call the original function, and by tagging each packet with a *reprocessing flag* that indicates if output should be produced.

The continuous processing on the migrate-from instance provides a source of current state information even if packets on the migrate-to instance are lost due to a buffer overflow. Additionally, the migrate-from instance keeps the state for some time after the migration is finished to ensure that relevant state information is still available for normal packet processing.

While the reprocessing of packets still requires buffers to temporarily store packets during migration, the buffers are removed from the critical path and thus packet processing speed is not decreased during migration.

The extended implementation reduces the controller load and thus significantly speeds up both the migration time as well as the packet processing during migration. It does, however, neither remove the limitations of OpenNF to only migrate state information between instances of the exact same implementation nor add the missing ability to share state information between different types of NFs.

2.5.8 *Stateless Network Functions*

Kablan et al. [34] introduce the concept of Stateless Network Functions (StatelessNF) where the functionality of a NF is decoupled from its state. While previous work focused on identifying and migrating relevant flow state, the authors of StatelessNF argue that the state of a NF needs to be cleanly separated from the actual operation and persisted in a backend store that can be concurrently accessed by all instances of the NF.

The architecture of StatelessNF currently consists of a processing tier and a data store tier but could be easily extended to additional tiers that process information in the background or provide security services for accessing the stored data. This dynamic architecture allows NFs implemented using StatelessNF to scale even on high load as the number of processing tiers can be increased without limitations that are otherwise imposed by state migration such as the need to redistribute flow state across new instances.

StatelessNF uses RAMCloud [54], a backend providing a generalized data store for datacenter applications. It runs on a cluster of storage nodes and provides a unified frontend to applications, managing data replication among connected servers and assuring data availability even on node failure. RAMCloud keeps the data in DRAM at all times and thus achieves very low latency both on read and write operations. The communication to the external data store is done using an InfiniBand based low latency network.

The authors created a dedicated RAMCloud client implemented as a library that acts as a layer between the NF and RAMCloud. The library handles both the communication setup with RAMCloud using IP and the actual exchange of information using DMA, thus bypassing the OS for higher performance.

StatelessNF provides both a blocking and a asynchronous (non-blocking) interface to the storage backend. Using the blocking mode requires only minimal changes to the NF as the calls to local lookups are simply replaced by calls to the external memory, thus blocking the execution until the network call to RAMCloud returns. While blocking execution to wait for the response of a local function or memory call is reasonable, using this method to access remote resources might lead to a decreased forwarding performance of the NF. Using the asynchronous interface circumvents this execution lock but requires additional changes to the implementation where the functionality must be split into a *pre-lookup*, *lookup* and *post-lookup* phase.

The authors also implemented a NF-local cache that holds the most recently read state information locally and thus avoids calls to RAMCloud.

To ensure data consistency, each instance of a NF acts as a liable source of state information for the flows it is assigned to. This guarantees that updated information is only actively used by the NF once it is persisted by the storage backend and thus the stored state in the backend always represents the current operational state of the NF group. It is not necessary for the NF to wait for the state information to be replicated in the storage cluster until it can be actively used.

StatelessNF focuses on the clean separation between functionality and state while providing near native access time to the state information. However, the externalized state is still specific to a certain NF implementation and thus can only be shared between instances of the exact same type.

Additionally, StatelessNF explicitly does not provide methods to annotate state but views the stored information as a *unstructured block of data* without any prescribed structure or schema. Furthermore, StatelessNF provides no means to persist and load snapshots of (partial) state at a specific point in time.

2.6 DISCUSSION AND SUMMARY

The systems surveyed in this chapter all provide means to share, migrate or replicate state information of different network components but with different use cases in mind. We provide an overview of the discussed systems in Table 2.2.

	STATE EXCHANGE		FEATURES			USE CASE			
	DIRECT SHARING	MIGRATION	PERSISTENCE	TRANSFORMATIONS	DECOUPLED STATE	NFV	NETWORK MGMT.	BGP	WMN
RCP [23]	+	−	+	−−	o	−−	o	++	−−
SDX [30]	+	o	o	o	+	o	+	++	−−
Statesman [68]	+	−−	++	−	++	−	++	o	−−
MobileMan [19]	++	o	−	−−	+	−−	−−	−−	++
CrossTalk [76]	++	o	−	−−	+	−	o	−	++
CLiSuite [44]	++	+	−	+	o	−−	−−	−−	++
VROOM [73]	−−	+	−	−−	−−	+	−	++	−−
Router Grafting [37]	−−	++	−	+	−−	o	−−	++	−−
SITN [71]	−−	+	−−	−−	−−	−−	−−	++	−−
Remus [20]	o	−	−−	−−	−−	+	o	+	−−
Tardigrade [46]	−−	+	−−	−−	−−	+	o	+	−−
Split/Merge [57]	−−	+	−−	−−	−−	++	−	++	−−
Pico Replication [56]	+	o	−−	−−	−−	++	−	o	−−
OpenNF [28]	−−	+	o	o	−−	++	−	+	−−
DiST [40]	−	++	−−	−−	−−	++	o	+	−−
P2P OpenNF [27]	−	++	−−	−−	−−	++	o	+	−−
StatelessNF [34]	++	−	o	−−	++	++	o	++	−−

Table 2.2: Overview of the related work. We use ++ (extended support), + (full support), o (partial support), − (rudimentary support), and −− (no support) to denote the feature set of an approach and its suitability for the use cases presented in Section 3.1.

Architectural changes in the network design have mainly be introduced with the usage of SDNs. The clean partitioning of network operations into management, control and data

plane as shown in Figure 2.11 has introduced a separation of concerns where decisions are taken in a central controller and the network hardware only acts as the execution instance of a centrally provided ruleset. This centralization has naturally led to the exchange of state information on the management and control plane while systems on the data plane keep their state locally due to performance reasons. Systems like Statesman coordinate the requirements of different network operations on the management plane to impose a unified and consistent state on the network that prevents a performance degradation or even network splits while systems like RCP handle the central coordination of packet forwarding.

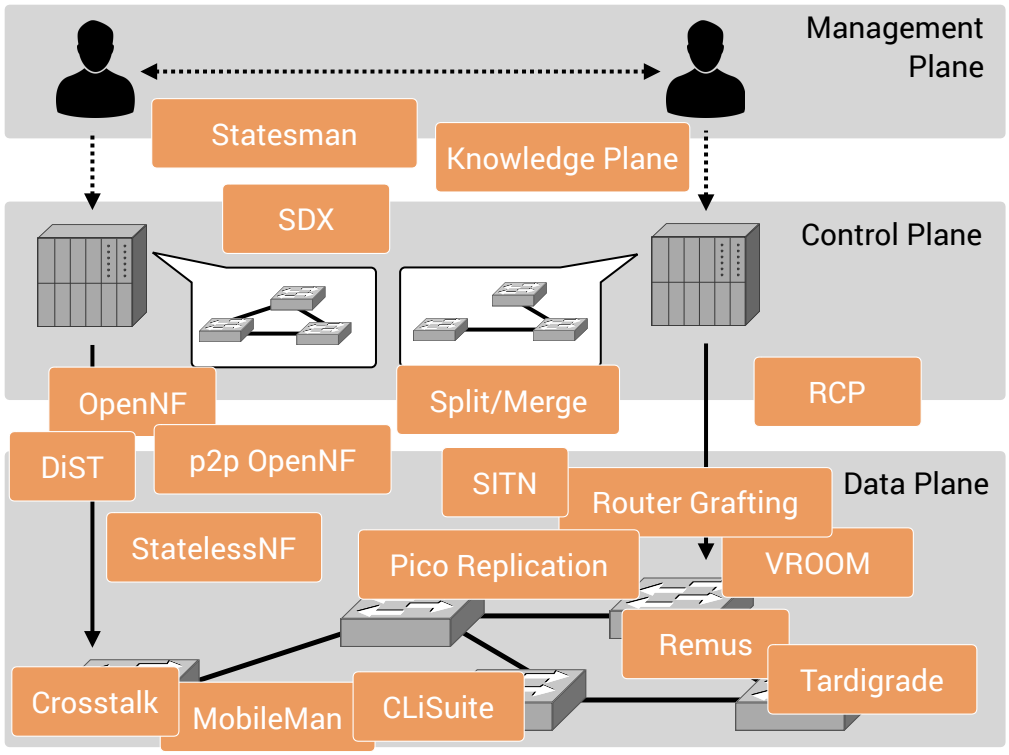

Figure 2.11: Related work and its position in a layered view of the networking architecture.

The integration of legacy mechanisms and the seamless migration towards a fully SDN enabled network is still one of the major challenges in this area [41].

On the one hand, architectural solutions such as SDX provide methods to unify the view of such hybrid networks by providing a common interface for both legacy as well as SDN-enabled devices. These interfaces are limited to the smallest common denominator when it comes to incorporating existing management planes.

On the other hand, cross layer solutions require the exchange of all legacy mechanisms in order to share state information. They remove the strict communication hierarchy by bypassing the layers of the network stack, and some frameworks such as CLiSuite even support rudimentary transformation functions. However, those approaches are focused on mechanisms running on the same node, and only extended architectures such as CrossTalk enable the sharing of information across multiple nodes. Thus, the use cases for cross layer systems are limited to specialized networks such as WMNs where each node runs independently, and bases its decissions only on locally gathered data.

In the area of NFV, some systems such as Remus and Tardigrade focus on the HA of networked systems, while the work on Split/Merge as well as OpenNF and its optimizations target the dynamic and efficient scaling of NFs during different load situations. The systems keep the state information as close to the functionality of the mechanism as possible, only extracting state during snapshot creation (in the HA use case) or during scaling events.

StatelessNF proposes a strict separation of functionality and state information leading to a new design principle when creating networked systems. This separation of state information has the advantage of using common methods to share information between different instances of the NF without the need to extract state from various locations beforehand. It usually requires a redesign of the software architecture as relevant state information now has to be identified and separated by the mechanism architect during the design process instead of leaving this task to a third party system.

We introduced the concept of a network context in Section 2.1. Existing work handles the sharing, migration or replication of current state representation using various technical solutions, but only few works even considers the other but equally important pieces of information. The configuration parameters are considered to be manually distributed among the various instances that use the same state information while metadata is not attached to the shared information, limiting the replication of state to the exact same implementation of the same mechanism.

Some systems such as StatelessNF even explicitly view the shared information as a *binary blob* without the need to know its exact structure or contents. Only Router Grafting considers the migration of state between routers of different vendors proposing an offline procedure to convert the data format between these mechanisms.

Historical records to recreate a state at a certain point in time are not considered by any system. Thus, none of the systems support a roleback to a previous known working state once the current state is finally committed. While such a short term roleback might be sufficient at a first glance, it can be advantageous to load known to work states where a quick adaptation to dramatic changes in environmental conditions or traffic characteristics is required and built in mechanisms are either not capable of handling these changes or are simply too slow to react.

The HA solutions use a heartbeat to detect a complete failure of the active instance but do not provide extensive monitoring information. All other systems rely on external monitoring to provide the controller with information necessary to police and intervene their operations.

None of the existing solutions supports sharing of the complete set of information that is composing the network context. Especially the metadata that enables sharing of information across mechanisms and implementation borders is missing in all the surveyed work. However, the extensive sharing of the complete context information is required to achieve a more flexible and dynamic network management, and further boost the deployment of new and innovative services.

3 PROBLEM STATEMENT

In this chapter, we first introduce four use cases that are references throughout this thesis, and that serve as examples for possible deployment scenarios for the proposed architecture. The scenarios are chosen to represent a wide variety of challenging network designs from state of the art core network deployments over data center infrastructure to upcoming access networks using device-to-device (D2D) communication.

Second, we introduce the concept of switching between mechanisms, and thus enable networks to cover a large spectrum of environmental and operational conditions. We argue that a switch becomes necessary when conditions change, due to the fact that mechanisms are designed with a certain operation environment in mind. Thus, changes in this enviornment also require a change in mechanisms to provide optimal network performance.

Third, we introduce the sharing of network context as one of the core enablers for innovation in network architectures. While sharing of state is already available in a narrow set of applications, we extend the sharing of information beyond those silos and propose an extensive exchange of context between all mechanisms within a network.

3.1 USE CASES

In this section, we provide several motivating use cases that show why broader sharing of context is beneficial for networked systems and network operation in general. The use cases serve as examples where sharing of network context can be employed to gain either additional performance, security or operational safety, or where sharing information can evolve the network itself, e.g., in a SDN or when deploying NFs.

We not only focus on newly developed mechanisms that can be built with sharing in mind, but also include legacy deployments where context sharing was not a first order design principle. In particular, we show how networks can benefit from context sharing between NFs as well as legacy mechanisms such as BGP routers and routing protocols deployed in WMNs.

The use cases described in this section are used throughout the remainder of this thesis to illustrate the concepts introduced. Additionally, the use cases serve as scenarios for the evaluation to show the feasibility of our approach.

3.1.1 *Migrating Network Functions*

The migration or sharing of context is a core enabler of employing NFVs as shown in previous work such as [28, 34, 57]. Tight SLAs on the operation of networks often demand a low packet processing time of NFs running in the data path of these networks while over-provisioning the hardware to address possible load scenarios is prohibitively expensive.

The dynamic scaling of virtualized NFs allows an operator to provide enough forwarding capacity during times of high load while shutting down unneeded instances

of the NF when the network is not fully utilized, thus saving energy and operational costs. The dynamic allocation of computing resources even enables the operator to scale out some of the running NFs to cloud providers if the resources available locally are not sufficient to handle the current load as shown in Figure 3.1. Additionally, the usage of virtualization allows multiple different NFs to share resources which leads to a higher utilization of the available hardware.

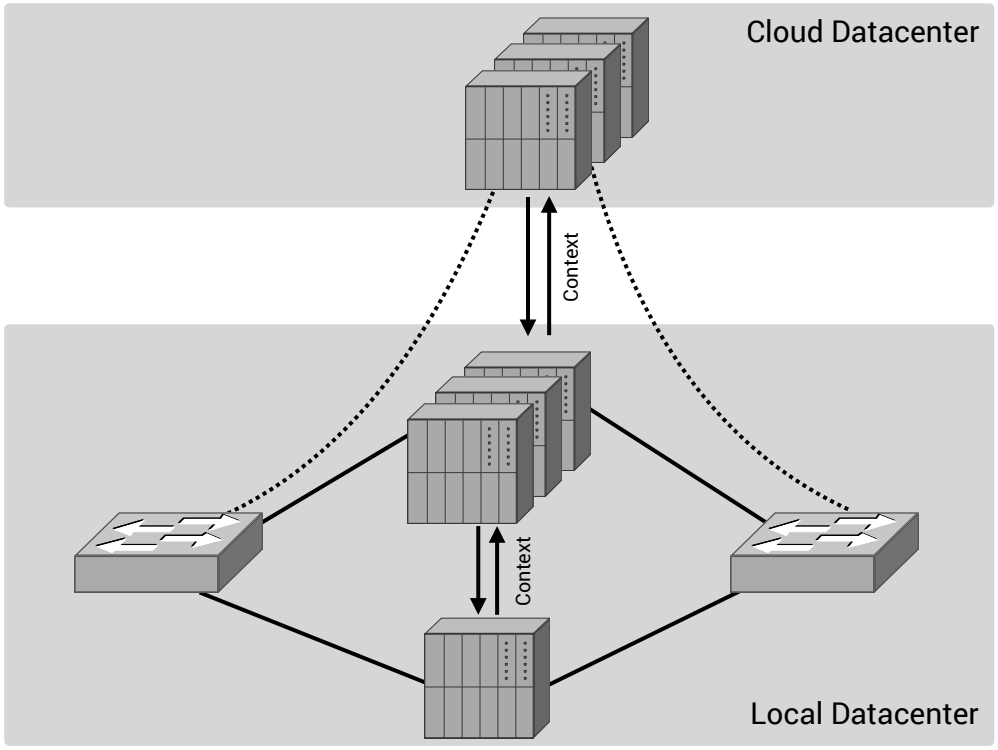

Figure 3.1: Dynamic scaling of NFs to the Cloud and context sharing between instances.

However, the dynamic scaling of NFs requires the underlying systems to either keep the network context consistent across all instances or migrate the relevant context together with the redirection of the corresponding flow.

For instance, an IDS stores state about each flow along with global counters and threat information gathered across all analyzed flows. Rerouting a flow to a new instance can significantly impact the detection accuracy due to missing context and can either trigger false alerts or leave attacks undetected. Thus, the sharing of IDS context improves the detection rate even when the mechanism is dynamically scaled and flows are redirected to other instances.

However, sharing context information is not only necessary for dynamic scaling of NFs. It is also required if an operator needs to replace an instance of a NF with either a version upgrade or with a new implementation. Especially networks that operate on tight SLAs which do not allow for service degradation or even outages require an in-place upgrade or replacement. Redirecting new flows to an upgraded instance and fading out the operation of the old instance when the handled flows are terminated is sufficient for networks with only short lasting flows.

However, only redirecting new flows is not reasonable when the network also carries long lasting flows that delay the fade out process for an undefined amount of time. The

migration of context between the current and the new instance helps reduce the total upgrade time as it enables the operator to redirect existing flows to the new mechanism as soon as the corresponding data is migrated without loosing information on the flow context.

Additionally, it might be necessary to employ specialized implementations of a NF for flows with certain characteristics. These specialized NFs usually need more resources, leading to a higher computational load on the hosts, to a higher delay in packet forwarding, and to a lower throughput. It is therefore not feasible to route all flows through this specialized mechanism as this would decrease the overall performance of the network. Instead, it is advisable to process all flows with a generic and high performance instance and to carefully select the flows to be processed by the specialized implementation. In the example above, a generic IDS inspects the complete traffic in a network segment and detects a suspicious flow. This flow is then redirected to a specialized IDS for further inspection.

3.1.2 *Reconfiguring Network Functions*

It is beneficial for services in a network to exchange relevant context information among each other and to cooperate in packet forwarding and manipulation in order to provide a higher service level. An asset management system like PRADS [24] might want to share the information about hosts and services in the network with an IDS to allow event to host/service correlation, or an IDS might wants to consent a firewall to access a list of malicious flows in order to block them as shown in Figure 3.2.

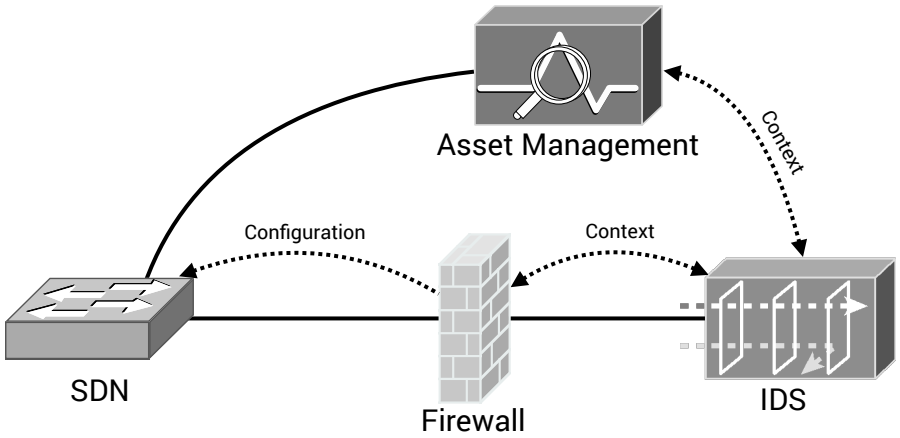

Figure 3.2: Reconfiguration of NFs dependent on the shared context. The SDN controller is omitted for clarity.

Another reason to share information between services in a network is the offloading of functionality to a better suited mechanism, thus reducing the load while still maintaining forwarding performance. For example, a firewall can offload filter rules to an SDN-enabled switch that processes packets at line speed. While advanced firewall policies and features such as connection tracking cannot be enforced on the switch, it is possible to execute simple filters based on IP addresses or TCP ports. The firewall engine continues to run on dedicated systems or as virtualized NF, but is able to extract rules and share this context information with the SDN controller that implements the policies on the switches.

Different groups of NFs usually run independently of each other as implementations are unable to directly share context with other mechanisms. Where sharing of context information exists, all information exchanged between groups of NFs has to flow via a central controller. This limits the possible sharing scenarios to a predefined set of information that is known to the providing and consuming NF as well as to the controller.

3.1.3 *Replacing BGP Routers*

Migrating BGP routers is a tedious task that involves multiple steps before a mechanism can be replaced safely. These mechanisms not only keep a RIB as context information but also hold long living TCP connections with all neighboring nodes as shown in Figure 3.3. These connections must not be interrupted as the neighboring routers will discard all routes learned from the node if the connection is lost.

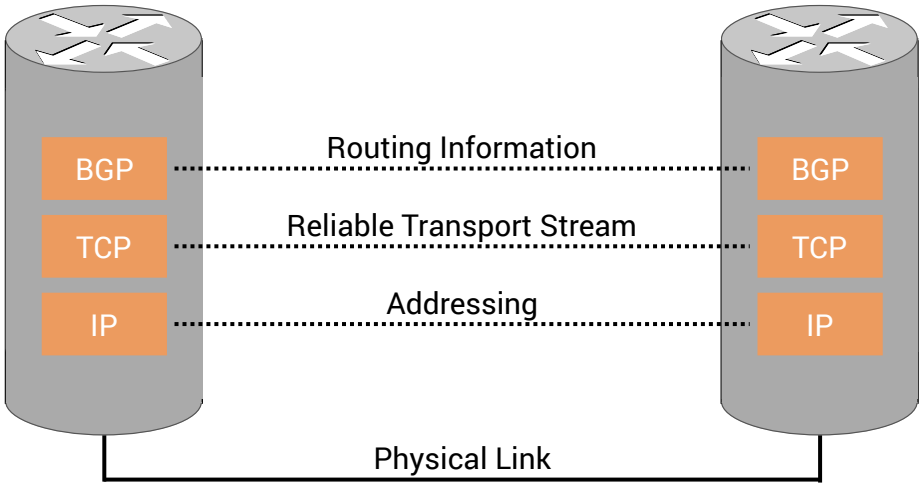

Figure 3.3: Layers involved in the migration of BGP routers. (adapted from [37])

Currently, the migration involves reconfiguring the routing protocols to route traffic away from the affected device, wait until the flow of traffic over the router stops, and then take the mechanism offline. This process can take tens of minutes and makes maintenance, dynamic adaptation, and load balancing of BGP routers hard [42]. Existing SLAs with customers also often prevent operators from taking routers down, or even from changing the internal routing without prior notice and without specifying a concrete maintenance window. These limitations often prevent the dynamic adjustment of internal routing as well as the on-demand replacement of router hardware, thus limiting the possible adjustments of the network to new traffic patterns or customer requirements.

Additionally, an operator might not want to migrate a complete router to another location but only move individual sessions from one device to another to relocate customers [5], or for traffic engineering purposes. This is especially important in SDNs where high dynamics in the network configuration exist and limitations of legacy mechanisms can hinder the evolvement of the complete network.

While migrating the BGP routing state between devices of the same vendor might be possible with proprietary solutions [17], the migration between routers with different implementations is a disruptive operation as each vendor uses its own representation of the routing state.

In conjunction with a SDN-enabled network, explicit support for sharing of routing context overcomes the vendor lock-in and provides capabilities for online conversion of context resulting in easier maintenance, faster adaptation, and better load balancing. This leads to a need for seamless migration of routers [37] where the BGP routing table as well as the TCP state and the IP configuration is moved from one device to another.

3.1.4 *Switching Routing Protocols*

D2D communication is a key technology in fifth generation mobile networks (5G networks) [53] to offload traffic from the Base Station (BS). The upcoming standards support a large variety of operational modes and parameters. The modes range from communication that is completely controlled by the infrastructure over central coordination but local traffic exchange to a complete decentralized network where devices coordinate themselves and only use some dedicated uplinks to the infrastructure. In these scenarios, routing protocols developed for WMNs are used to find neighboring devices and to route traffic through the network without central coordination or control.

Today, WMNs are already in operation where wired infrastructure is either not available or too expensive to deploy.

On the one hand, this can be rural areas [22] where low cost network access is essential for the education of children and the development of the area but the spending power of the people is very low. Connecting these neighborhoods by laying cables or deploying infrastructure based wireless technology such as cellular networks or IEEE 802.16 WiMAX is prohibitively expensive.

On the other hand, deploying wired technology to form the backbone of a wireless access network can be challenging even in splendidly constructed areas [3]. Government regulations and the populousness make wiring works difficult and limit the quick extension of the network. This is particularly true where a high density of access points is required.

Thus, WMNs can be used to form a wireless backbone to reduce the construction work required in both rural as well as in congested areas.

However, a variety of parameters and environmental conditions have to be considered when planning and deploying a WMN. The conditions and parameters can range from the population and usage density over the general weather situation to the expected usage scenario. Especially the latter is crucial for the selection and parameterization of the mechanisms used. A community network with free access for everyone requires fast network links but is expected to be unreliable, while a network built to support emergency services must be reliable but the bandwidth demands are likely lower.

These considerations result in a choice of mechanisms that are fixed over the lifetime of the network as changing or adapting the networking stack to varying conditions or usage patterns basically results in deploying a completely new mechanism configuration. This static deployment limits the usage of networks for multiple purposes and results in either multiple networks that serve special purposes and are not federated due to different mechanisms used, or in not deploying networks at all as specialized mechanisms that are only used very scarcely are too expensive to provision.

The dynamic adaptation of routing protocols provides a possible exit route to this dilemma. It allows operators to deploy WMNs running a mechanism that is designed for

the current usage scenario, but allows the conversion of the network—within the bounds of the deployed hardware—if other scenarios arise.

The mechanism switch within a WMN must be seamless without interruption of end to end connectivity and transparent to the end user. This is not only important for the rare case of a complete repurposing of the network but especially for the dynamic adaptation to changing conditions during runtime when the global usage scenario remains the same.

3.2 SWITCHING MECHANISMS

Mechanisms are usually designed for optimal operation in a narrow set of environmental and network conditions. These conditions can range from the number of concurrent users or flows in the network over the available bandwidth that can vary depending on the physical route of a transmission in a network to the environmental conditions such as the temperature and the precipitation that severely impact the lifetime of batteries and the propagation characteristics of wireless signals. Once the operational boundaries envisioned by the mechanism designer are left, the performance and efficiency usually decreases radically.

Thus, there is no one single mechanism that is able to fullfil the requirements imposed by all possible users, environments and network properties.

Switching between different mechanisms is one way out of the dilemma. Instead of deploying one mechanism that needs to fit all requirements and conditions that might occur—and are potentially not even known during deploy time—we implement and roll out multiple mechanisms that are each optimized for a narrow set of operational conditions.

Additionally, we implement a system that is able to coordinate and orchestrate the transition between mechanisms along with a monitoring and decision engine that allows us to observe the network behavior as well as the environmental conditions and select the most appropriate mechanism for the current situation out of the set of available mechanisms.

By enabling those mechanisms to be transition aware, we can ensure that mechanisms are able to announce their properties and operational parameters during deploy time and that the optimal mechanism for the current conditions is selected during operation.

There are two possible ways of switching mechanisms: 1. the hard switch where one mechanism is disabled immediately before another mechanism is engaged, and 2. the soft switch where multiple mechanisms run in parallel during the transition phase.

The hard switch uses only minimal resources on the hosts during the transition as it only requires the orchestrator to run during a short period of time, and the mechanisms do not consume any additional resources. However, the hard switch does not allow for a seamless transition—without external support—as the mechanisms are not executed in parallel and thus cannot set up their state to match the current environment. Thus, there will always be a gap in network traffic until the new mechanism is fully operational.

During a soft switch, multiple mechanisms—usually two—are operated in parallel. While the currently selected mechanism is still responsible for handling the live traffic, the new mechanism is receiving a copy of all network flows to establish its operational state. This enables the new mechanism to bootstrap its internal data structures using the

information available on the network node without requiring the mechanism to directly handle the data traffic.

One of the main challenges when switching mechanisms during runtime is finding a consensus between all participating nodes in a network that one or more mechanisms need to be exchanged. This is especially true in distributed systems such as WMNs where no central instance exists that can enforce the switch. We need to gather comprehensive monitoring data from all connected mechanisms as well as their environment to reach a concise decision on when to switch and what mechanism to choose next. This information has to be made available to all participating nodes in order to make informed decisions that are viable for all affected nodes in the network.

Additionally, when a consensus is reached and the decision for a switch is made, we need to orchestrate the transition process. This not only requires a tight monitoring to ensure that all nodes switch to the new mechanism and are using the selected parameters, but also a rollback strategy in case the selected transition path fails. We must ensure that no node is disconnected from the network due to the transition and that all affected nodes use the same set of new mechanisms and parameters for a seamless operation.

Designing mechanisms that are transition aware is the second challenge when switching. While it seams reasonable at first to simply deploy existing implementations and only add a thin additional layer to support the switch on the underlying network stack, this also implies that the mechanisms are not aware of the complete system behavior and cannot react to events initialized by the transition infrastructure. Even when only using a hard switch to change mechanisms, it is usually feasible to not completely shut down the running mechanism but to introduce a *hibernate* state where computational intensive and network traffic generating operations such as timers or responding to connection requests are disabled but the program code is still held memory. This enables operators to switch between mechanisms within a couple of milliseconds and thus utilize transitions to adapt the network to highly dynamic demands.

The introduction of a soft switch requires further adaptations of the mechanisms. In addition to the dynamic control of timers and operational state, it also requires fine grained control over the output of a mechanism to not duplicate the packets during the bootstrap phase while still being able to send and receive control traffic necessary to fully create the operational state. Thus, it is also not feasible anymore to filter, select and redirect the packets at the interface level but all information has to be forwarded to transition aware mechanisms that can base the handling decisions on their internal state, current operational status and their requirements.

During a soft switch, we also need to determine the time required for parallel operation to ensure that the new mechanism has created all state required for its seamless operation while not wasting additional resources due to an unnecessary long parallel execution of both mechanisms. This transition time is dynamic and largely depends on the current network utilization along with the operational conditions of the nodes involved in the transition, but also on the selected mechanisms and parameters.

Therefore, we need to take the required transition time into consideration 1. when deciding which mechanism to choose and selecting the transition path to ensure that the transition does not take longer than the envisioned operation time, and 2. during the execution of the switch where the parallel execution needs to run as long as necessary and as short as possible to provide a seamless transition.

The requirements for more dynamic networking structures and mechanisms demand not only flexible forwarding infrastructures and algorithms on the packet level using SDN but a dynamic adaptation of mechanisms on all layers of the network stack. Thus enabling transitions between mechanisms of the same class is a route towards the required flexible network behavior. However, enabling mechanisms to be transition aware introduces some additional challenges from the decision when to switch and where to switch to, over the orchestration of the transition process to the mode of operation during the transition phase.

While we recognize the importance of the decision and orchestration, we focus on the requirements and modifications necessary to design, implement and deploy transition aware mechanisms, and their behavior during the switching phase.

3.3 SHARING CONTEXT INFORMATION

The sharing of network context is not only a key enabler for a more dynamic network management but also a possible solution to reduce the time of parallel operation in a soft switch scenario as described in Section 3.2. The exchange of context information between mechanisms fosters innovation in all fields of network deployment and operations from the fundamental services such as BGP over state of the art architecture concepts like SDN and NFV to mechanisms that are designed for special environments or usage scenarios such as WMNs.

Thus, context sharing leads to a better utilization of existing resources, higher throughput, lower latency and optimized network operations, resulting in superior adherence to SLAs and an improved user experience.

Additionally, the extensive sharing of context information allows for additional and new functionality that is not available when each mechanism is operated by itself and is not aware of the current operational parameters, other mechanisms in the same network segment are running on.

Context can be shared in two distinct ways: 1. the migration of information between mechanisms at distinct events such as a switch in mechanisms, and 2. the continuous exchange of context during all stages of the mechanism life cycle.

Using context migration, each mechanism is responsible for managing its context during normal operation and needs to provide data structures as well as functionality to store and control the usage of the information.

On the one hand, this allows the mechanism designer to use patterns optimized for the current implementation and the context handling of existing implementations does not need to be altered. The complete local handling of context information allows for native performance as the information is handled in native data structures and the access patterns can be optimized to suite the features of the programming language as well as the features of the runtime environment used.

On the other hand, it requires functionality for context handling in each mechanism design and synergy effects for context management are only minimal as each mechanism runs its specific implementation. It limits the exchange of context between different implementations or even different types of mechanisms as the context is highly specific to the source mechanism and the possibilities of integrating context transformations are limited.

The local context handling usually also does not include the supply of monitoring data nor does it allow for the generation of a history or the insertion of previously gathered context. Therefore, it only allows for a limited set of information that rudimentary conforms to our definition in Section 2.1.

Additionally, the migration of context requires an API that allows to extract the context information from one instance of the mechanism and inject this information into another. This interface needs to be explicitly supported by the mechanism developer and therefore the migration support introduces additional overhead alongside the local management of context.

While the migration of context usually does not require an external system to handle the information but implements a direct exchange between the involved mechanisms, an external controller is necessary to schedule the information transfer, control the migration path and orchestrate the exchange.

The continuous sharing of context enables mechanisms to access information of other connected mechanisms during the complete lifecycle and therefore constantly adapt their operation to the current network status without the need to wait for new information during a migration cycle. This allows for highly dynamic networks without the overhead of triggering context migrations at a high frequency.

Using continuous sharing, the context is best managed in an external system that is solely responsible for handling the information provided by the various mechanisms available in a network segment and that controls the lifecycle of the stored data. The context management system relieves the mechanism developer from implementing a custom context handling and allows to hand of all context management by providing a flexible API that enables a standardized way of organizing the context information across various mechanisms and removes the need for customized solutions.

The management system provides access to the complete context history of the network segment and allows operators to infix previously gathered information into the mechanism when necessary. It is also able to provide preprocessed information for monitoring systems that require context information to gain a complete view of the network and the connected mechanisms.

The continuous sharing of information using an external context management system enables the exchange of information between different mechanism implementations as well as across mechanism classes. The centrally gathered information is available to all connected mechanisms and thus the extensive sharing of context is facilitated, leading to more information available at each mechanism. This additional data can be used to further optimize the operation and allows mechanisms to cooperate within their network segment.

However, the integration of an external system comes at an increased latency to access the context information and therefore a potentially lower performance of the mechanisms. This can be circumvented by using local caches within the mechanism implementation which reduce the number of requests to the external system required while still keeping the context information current.

Both methods of sharing context information require a granular selection of what information needs to be shared. While it seems obvious to simply share the complete set of context information between mechanisms, the overhead increases with each exchanged data point without necessarily increasing the benefits. Caching information for instance is taking a large amount of context storage in certain applications but can be easily

recreated. Another example is short lived information that is already outdated before the exchange between two mechanisms is completed. This could be for instance the physical layer properties of a wireless channel or the memory page information in an OS.

The exchange of context information is a complex and challenging task not only in terms of selecting the most suitable sharing strategy depending on the mechanisms and the goals pursued but also in terms of choosing the information that needs to be shared.

While the former decision is usually taken by the network operator depending on the current mode of operation, the selection of relevant context can only be done by the mechanism designer who has specific knowledge on the internal operations of the mechanism.

The migration of context at distinct events is usually used for failover scenarios and scaling requirements where mechanisms run independently of each other, and an external system monitors the operational environment and triggers the migration of context information if predefined thresholds are exceeded.

In this thesis however, we focus on enabling a continuous exchange of context information in conjunction with sharing information across multiple mechanisms to allow operators to design and built more dynamic networks.

3.4 CHALLENGES

The exchange of context information is technically challenging for three main reasons: First, context information may not always be stored in equivalent form, resulting in the need of translating state information during the transition phase. Second, many implementations are of rather monolithic nature; however, context transfers between instances require a clear encapsulation of the mechanism context and according interfaces: if the mechanism context is dispersed all over an implementation, it is very hard to extract and to inject the context if a transition is fired. Third, it must be ensured that the context transfer between the implementations does not yield leakage of sensitive or private data.

However, these challenges are not only directly imposed by the mechanisms that access the context management but also by the circumstance that the information is now shared among mechanisms.

In this section, we break down the demands of the targeted mechanisms and the use cases that go beyond the requirements of a stand-alone implementation and focus on the extended set of requirements that comes with the introduction of an external context management system. We identify, extract and generalize the challenges that are imposed on a generic system specifically designed to manage network context.

3.4.1 *Enable Seamless Migrations*

The most fundamental issue is that changing or migrating a service forces context information to be newly negotiated or reestablished—requiring setup time—before the new service is operational. Transitions can result in service gaps during which no data can be processed or forwarded due to missing context. Thus, we must reduce the service downtime by sharing established context information between the different mechanisms.

We need to either migrate the context information before the link migration is initiated or externalize the context storage to a separate and specialized system that enables concurrent access for all mechanisms.

On the one hand, the on-demand sharing allows the mechanisms to store the context in local data structures and enables direct access, it requires the sharing system to either pre-populate the data structures before the migration is initialized, or—in case of a failover scenario—to constantly update the information on the passive or stand-by mechanism.

A central context management on the other hand enables mechanisms to directly access all shared information without the need for an external system to update internal and private data structures.

However, context transfers between mechanisms require a clear encapsulation of the context and according interfaces: if the context information is cluttered all over a mechanism implementation, it is very hard to extract and to inject if a transition is fired. The support for seamless migration thus requires the mechanism designer to gather all relevant context information at a central point and clearly separate the functionality from the context.

3.4.2 *Provide a Generic Context Storage Solution*

The context management system must provide a general storage solution for all connected mechanisms as it can be prohibitively expensive to customize already existing implementations to suite a new system. This does not only include the support for data types that are used but also for the metadata that is usually assigned to each value such as the storage and last access time, the visibility to other instances, and the expiration.

However, the support for an extensive set of data types and the associated metadata is not only limited to the storage backend. Additionally, the context management system must provide an expressive interface that is used by the connected mechanisms to store and retrieve the context information as well as alter the information associated to it. This includes a query language that supports the dynamic composition of requests and allows for dynamic filtering depending on the desired information.

3.4.3 *Provide Context Persistence*

The network context does not only include the current configuration, state information and metadata but also comprises historical records of these information as described in Section 2.1.

We thus must prevent the loss of context due to a system reconfiguration or failure, and offer access to the historical records along with the possibility to easily restore a previous state. It is not sufficient to hold the currently valid information in memory but we must provide means to persist the information held by the context management system, and to load this saved context information at a later point in time. The persistence of context information also allows us to later load the previously known context that best matches the current environment if a change in the operating conditions or operational requirements occurs, and the new conditions were experienced before.

Additionally, the persistence of context information allows offline operations of the collected information by duplicating the stored data to another system. This enables us to run resource intensive tasks on a additional copy of the context instead of directly accessing the context management system that continuously processes the requests of the connected mechanisms.

In order to efficiently support historic information as required by the network context, the storage also needs to be able to generate a consistent and self contained snapshot of stored context information at a certain point in time. This requires that the context management system is brought to a stable state where all dependent operations are finished before the snapshot is taken, and that the resulting snapshot does not only contain the context information currently governed by the system but also the configuration parameter and internal state of the context management system itself. Only the complete set of data ensures that the snapshot can be restored later and the information loaded into the storage is consistent even when used across multiple instances.

3.4.4 *Reduce Overhead*

Network mechanisms such as NFs and routers are required to process a large amount of packets, each resulting in a context lookup or even update to the context management system. Thus, the overhead introduced by the system must be minimal to not reduce the overall performance of the network by introducing context management, and to satisfy even tight SLAs.

The context management system must not only be able to handle a large amount of concurrent requests, but also process these requests with 1. a low latency, and 2. low resource consumptions on the host system. While the former is necessary to support the processing of packets for real time applications and in high speed networks such as the Internet core, the later is especially relevant if the context management system is running on a low powered device that is executing other processes and also hosts the accessing mechanisms. This could be for example a WMN node running the context management along with the routing protocols that are storing and retrieving information to forward packets to other nodes in the network. Offloading the context management to a secondary system is not possible in this scenario as the outside connectivity directly depends on the availability of the context.

We must not only confine the computational and network overhead but also the memory consumption. The context management is required to store a large amount of data and thus the available memory for both active as well as persistent storage must be used efficiently. While state information is usually only small compared to the information that is processed by the mechanisms, the historical records can—depending on the snapshot frequency—significantly increase the size of the required storage.

3.4.5 *Enable Dynamic Reconfiguration*

The context management system is responsible for storing and providing the core information to the connected mechanisms which are not operational if the information is delayed or missing.

Therefore, we must provide a high availability and minimize outages due to system maintenance. The support for dynamic reconfigurations of the context management system is an essential feature to support when new mechanisms are introduced or the parameters of the system itself need to be adjusted.

The context management system is designed to support arbitrary mechanisms and it is usually not known in advance which mechanisms are connected during the life cycle of the instance. Thus, we need to provide a system to dynamically load new or changed

functions if connecting mechanisms require additional or altered data transformation functions to operate on already gathered information.

The newly loaded functions need to be announced within the context management system, and the system therefore must provide a configuration API that enables the operator to register new mechanisms as well as additional features such as new data transformations during runtime. Additionally, the API must allow to set the operational parameters of the context management during runtime and provide endpoints to completely reconfigure the instance.

3.4.6 *Support Data Transformations*

The use cases require the sharing and re-use of context information across different mechanism implementations or even across different application domains. This requires the possibility to adapt the information provided by the context management to the needs of the accessing mechanism in both data representation and semantics.

While it might be tempting to adapt the internal state representation of each mechanism and accommodate the implementation to a generic format, this leads to a high complexity of the required adjustments as the functionality of a mechanism is usually closely coupled to the state representation. The changes applied to the data format also require modifications of the underlying functionality and can even lead to inevitable changes of the externally visible functionality or features.

Additionally, when designing a common format for storing the context information, all possible current and future mechanisms, including their requirements and specifications, need to be considered. However, this is usually not feasible as networks are subject to constant change as new mechanisms are introduced and existing solutions are decommissioned. The constant change in network configuration and the used NFs would lead to regularly re-specifying the base context and then adjusting all implementations to the new base context.

Therefore, it is not feasible to adapt the connected mechanisms to a central context management by specifying a global and universally valid base context, and modifying the implementation of each mechanism to match this specification.

Instead, the modifications required to connect a mechanism to the context management system must be minimal, and the connected mechanisms must be able to share information without agreeing on a common context. Hence, a mechanism might profit from the information others have contributed without being explicitly aware of the existence, or even the context, of other mechanisms.

We need to implement *data transformations* that allows us to 1. adapt the information provided by the mechanism to the internal representation of the context management upon storage, 2. convert the data format of the stored information to match the representation required by the requesting mechanism during retrieval, and 3. retrieve semantically equivalent information either directly from the context storage or with the support of external systems when the requested data is not directly available. The operations can range from simple type conversions over mathematical functions such as the average over a series of values to complex methods with external calls such as the conversion from an IP address to a Media Access Control (MAC) address employing an Address Resolution Protocol (ARP) lookup.

This support for data transformations is the core enabler for sharing context between different network components.

3.5 SUMMARY

In this chapter, we introduced the challenges that arise when sharing network context across multiple mechanisms along with the requirements for introducing a system that supports transitions in various networking environments.

We introduced multiple use cases that represent the diverse set of mechanisms and deployment scenarios that can occur in a heterogenous network environment and that all benefit from sharing context information. Our use cases range from core network services such as BGP where we show how a seamless replacement of BGP routers can be implemented using context sharing, over the migration of NFs that allows for HA applications and the fulfillment of tight SLAs to the introduction of extended sharing of context information to reconfigure NFs. We also target the transition between different mechanisms of the same class. The use case describes how switching WMN routing protcols can be realized to adapt networks to different environmental or operational conditions.

We extracted two major problem domains from these use cases, namely: 1. switching mechanisms during runtime, and 2. the implications of sharing context information.

The switching of mechanisms is required for an optimal operation in changing environments as mechanisms are designed for a narrow set of conditions. We argue that switching between multiple mechanisms leads to a better network performance during the complete lifecycle of the deployment and show what challenges arise when such a dynamic transition is introduced. While we recognize the importance of the network wide support systems such as monitoring, decision and execution of the transition, we specifically focus on the design and implementation challenges for transition aware networks on a mechanism level.

The exchange of context information supports operators in optimizing existing networks and enables network architects to design new mechanisms dependent on information already available in the network. We stated that the extensive sharing of network context is critical to facilitate development in SDN as well as existing infrastructure such as BGP routers. We showed the challenges that arise from sharing information not only between mechanisms of the same implementation but between different classes of mechanisms.

Finally, we presented the demands of the targeted mechanisms that go beyond the requirements of a stand-alone implementation, and keep a focus on the introduction of an external context management system and the extended set of requirements that comes with its introduction.

4

SYSTEM ARCHITECTURE

Our survey of the use cases and the challenges that arrise from those scenarios presented in Chapter 3 show that the problem of enabling transitions between mechanisms in a network is twofold:

First, we need to introduce a state plane that is able to handle the network context of all mechanisms deployed in the network. We must 1. provide a central storage for the participating mechanisms to store their context information in a structured format, 2. enable the retrieval of context that is either provided by the requesting mechanism itself or by another mechanism, and 3. provide transformation functions to enable connected mechanisms to share information without agreeing on a common context.

The state plane must also provide means to persist and load context information depending on the current environmental and network conditions as well as take regular snapshots of the information to hold a history of the context.

Additionally, it requires interfaces to external systems such as the monitoring infrastructure to supply the transition architecture with relevant information and ease the decision of initiating a mechanism switch.

In Section 4.1, we present the design and implementation of STEAN—a Storage and Transformation Engine for Advanced Networking context. It provides an architecture for a context management system specifically developed to support a wide range of network mechanisms such as NFs, WMN routing protocols and BGP routers, and acts as a state plane to handle all relevant context in a specific network segment.

STEAN is designed to handle both legacy mechanisms as well as mechanisms that are designed and implemented with transitions in mind. Thus, it is the core system that enables the extensive sharing of context information between mechanisms, and allows for a more dynamic network management.

Second, we need to provide a system architecture that 1. allows us to easily extend existing mechanisms to support transitions, 2. enables software designers to develop new mechanisms that are specifically created with transitions in mind, and 3. supports operators to deploy those mechanisms in both existing and newly rolled out networks.

Additionally, we need to ensure that a synchronous transition between the services is supported by all participating nodes, and we must provide methods to coordinate and control the switch within the network to not disconnect sub-networks during or after the transition. We present a blueprint for such an architecture in Section 4.2.

Our architecture provides a paradigm shift, such that it takes transitions as a first order principle. The modular nature of the architecture allows us to adapt the components to the requirements of a highly dynamic network, and optimize mechanisms depending on the specific use case without the need for a complete reimplementation.

4.1 STEAN—A STORAGE AND TRANSFORMATION ENGINE FOR ADVANCED NETWORKING CONTEXT

The introduction of a state plane is the core enabler to a wide sharing of network context between all mechanisms running in a network and across all layers of the network stack. This state plane is responsible for managing all context information gathered by the various mechanisms, resolves any conflicts in context data, and provides the collected information to other mechanisms that benefit from the additional information. It also provides information for monitoring systems that are able to directly gather data within the mechanisms instead of relying solely on information that is observed from the outside, and keeps historical records to return to a known previous context if required.

In this section we present the *Storage and Transformation Engine for Advanced Networking context (STEAN)*, an architecture for such a state plane along with an implementation of our design to validate the concept. The presented system provides a generalized way to share context in a diverse set of core functionality such as routing, network processing, and dynamic mechanism adaptation without loosing the particular quirks of each context representation that is required for an efficient operation of the mechanism. This relies on collecting and managing context from different mechanisms using their preferred context representation, thus replacing per-entity state storage with a shared context management. STEAN makes this dynamic information available to other mechanisms, and stores and persists the current context for re-use at later points in time. This way, our system enables any component in a network to access other components' context information, hence facilitating seamless network transitions. It aids in seamlessly changing between mechanisms at runtime, or access persisted information describing the mechanism context from the past to reach fully operational state with minimal latency.

We also introduce transformation functions that allow for context sharing between mechanisms that where not originally build with sharing in mind. They enable mechanisms to re-use context from other mechanisms without agreeing on a common state representation, and thus allow STEAN to be integrated into legacy mechanisms and to interoperate with arbitrary mechanisms, which permits the seamless extension of existing stacks and network topologies.

Furthermore, transformation functions allow us to share context between different NFs that are—until now—only designed to exchange state between instances of the same implementation. The STEAN architecture is built with transformation support as a first order design principle and transformation functions are at the core of our implementation. This ensures that the overhead introduced by the transformations is kept to a minimum and all other features are optimized towards an efficient processing of transformations.

Additionally, we show how the state plane can be integrated into newly designed mechanisms as well as legacy mechanisms, and provide client implementations for a wide variety of network mechanisms from WMN routing protocols to SDNs. The presented library supports multiple platforms from modular mechanism frameworks such as Click to control plane architectures for NFs such as OpenNF. We consider each connected mechanism that is able to contribute information to the context store as well as request information from the context management system a *client* to STEAN regardless of its functionality or implementation.

By deploying STEAN as a state plane, network operators not only benefit from the extended sharing of network context and thus from improved operational parameters

but the benefits also result from a faster development and deployment cycle. Mechanism architects do not need to design and implement individual state storage and management solutions but can focus on providing the desired functionality while using a standardized API for state access and leaving the context handling to STEAN.

We unify the strength of the current systems by removing their individual weaknesses, and we argue that the gains of a unified system—with respect to support context sharing—are higher than the possible performance losses introduced. A general analysis of the performance is presented in Section 4.1.8, and the results of our evaluation—based on multiple use cases—are discussed in Chapter 5.

4.1.1 *Assumptions*

The proposed system offers functionality for context transformation, storage and retrieval in both legacy and SDN-enabled networks using wired as well as wireless technologies. Our goal is to support context sharing among all mechanisms connected to a network and even across network boundaries to fertilize the innovation in the network and support the transition towards a flexible and dynamic network management.

While we target a wide variety of deployment options, we base the presented work on the following assumptions about the design of the network infrastructure, the connected mechanisms and the deployment of STEAN:

4.1.1.1 *One STEAN instance per functional group*

STEAN is designed as a centralized context management system that forms a state plane for the complete network infrastructure. However, while it is usually desirable to have one (logically) centralized system that handles all context information in an entire network, it is not feasible to deploy just a single instance where potentially thousands of mechanisms contribute information.

Thus, we follow the same approach as in SDN-enabled networks where one logically centralized controller is managing the complete network but the actual deployment follows a distributed approach. We assume that each logical functional unit in a network—such as one node in case of a WMN or a functional group of NFs—contribute information to the same STEAN instance that is placed close to the requesting mechanisms. This not only eliminates the single point of failure introduced otherwise, but also reduces the latency of requests as the systems responsible for a certain slice of the network can be placed closer to the managed mechanisms.

4.1.1.2 *Separation of functionality and context*

A modular design and clean separation between the functionality and the context of the deployed mechanism is critical for the network wide sharing of context information as discussed in Section 4.2. While we are aware of the fact that most existing mechanisms are not designed with sharing in mind and the modification of existing implementations is a tedious task, the change in design principles and the adaptation of already deployed mechanisms is crucial to the successful introduction of a state plane.

Therefore, we assume that the connecting mechanisms are either directly developed following the presented design principle or are already modified to extract the relevant context using frameworks such as OpenNF. The modifications necessary to provide the

information can also be supported by using a tool such as StateAlyzr [38] that automates the process of identifying relevant state information.

4.1.1.3 *External trigger for functional migration*

Two of the core use cases, namely the migration of NFs and the switching of routing protocols, require a monitoring and decision system that continuously supervises the network and orchestrates the migration between mechanisms. Various systems that are specifically designed to handle the migration of NFs already exist, and we sketched the requirements and possible design choices for such a system to be deployed in WMNs in Section 4.2.1.

In the remainder of this section, we assume the existence of a transition controller wherever necessary and solely focus on the sharing of context information between mechanisms that are orchestrated by such a system.

4.1.2 *Base Context*

STEAN uses a context representation that—in conjunction with the transformations introduced in Section 4.1.3—allows network architects and operators to combine the required information of various connected mechanisms into a consistent model. This unified context representation—we call the *base context*—removes the need to store duplicate information contributed in different formats and fosters the sharing of context between mechanisms.

Despite the fact that we unify the context handling throughout a network segment or mechanism group, the base context still strongly depends on the mechanisms deployed within the specific network. Thus, it needs to be adapted to the current deployment situation, and the design of the base context must include possible future extensions of the network. Thus, we are not able to provide a single unified model of the base context, but show in Section 4.1.7 how the different base context for our use cases are designed and which information is matched between the mechanism implementations.

The construction of the base context is currently done by the network architect or operator during the intial deployment of STEAN. The adaptation to the available mechanisms requires a deep understanding of the network architecture as well as the operational mode of the connected mechansisms. The design and implementation of the base context is done in close cooperation with the design of the context transformations as those two elements directly relate to each other.

The base context uses labels or tags—we call them annotations—to address the different entries. Those annotations are definied within the base context itself, and mechanisms can neither add nor alter annotations but have to use the set provided by the context management system within the current base context. This allows us to further abstract from the mechanism specific context, and forces the consistent use of annotations throughout all mechanisms.

The architecture of using an adapted base context for each mechanism group allows us to unify the context of those mechanisms without loosing any information due to either an incomplete data model or a high level of abstraction that tries to unify all possible scenarios in one common base context. Wherever the projection of future mechanisms

within the network is not possible, our concept also allows for migrations between two base contexts to enable future changes using context transformations.

4.1.3 *Context Transformation*

The support for context transformations is the core enabler for sharing context between different network components. STEAN implements transformation functions that enable connected clients to share information without agreeing on a common context. Hence, a client might profit from the information others have contributed without being explicitly aware of the existence, or even the context, of other mechanisms.

We use a running example throughout this section to show how transformation functions can be employed to share context between independent NFs: a network consisting of a NAT, an SDN-enabled switch that balances the traffic load between two firewalls, and an IDS (Figure 4.1). All NFs in this example are STEAN-enabled. The SDN controller providing rules for the switch is connected to STEAN, where it stores its state, including the SDN rules. During normal operation, the firewall instances share their context using STEAN and the other NFs operate independently of each other, using STEAN as their context storage.

Now, we consider the following failover scenario: Link 1 carrying the traffic assigned to Firewall A fails so all traffic is re-routed to Firewall B. The traffic load exceeds the capacity of a single firewall instance, thus traffic must either be dropped or SDN rules must be dynamically generated to enable a pre-filtering on the switch.

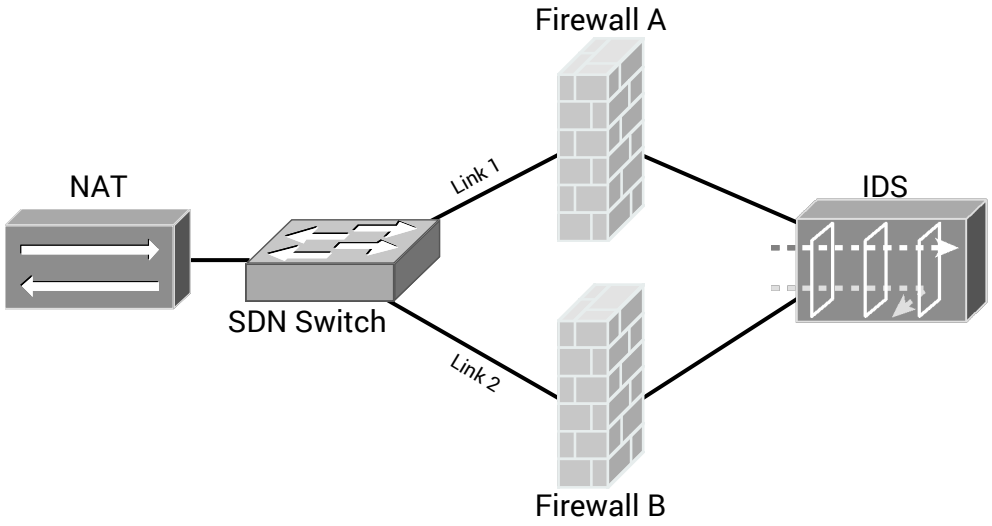

Figure 4.1: Simple network to show the advantages of sharing context between different NFs. All NFs as well as the SDN controller are connected to a central STEAN instance (omitted for clarity).

4.1.3.1 *Concept*

Transformations allow developers to create and use an extensible set of functions that acts as an additional layer between the client and the context storage. This layer is responsible for translating between the client-specific context and the common base context. It allows the client to store and retrieve the information “as is” and “as needed” without adapting

its internal representation to the one used in the context management system. The client does not need any information on how the context of other clients has to be interpreted. This interpretation is provided by the transformation functions that offer a client specific view on the base context.

In our example, the NFs can share state between each other to allow for a flexible load balancing and dynamic reconfiguration in case of failure. Each NF as well as the SDN controller keep their internal state representation, and STEAN provides transformation functions for each client. We identified four different types of transformation functions that enable different operations on the stored context:

1. *Filter Functions* are applied during data retrieval and limit the results to the context information that is relevant to the client. The storage system iterates over the data items during the lookup process and applies the filter functions on each item or until a specified amount of matching results has been found. The filters are applied as early as possible. This minimizes the amount of data passed to subsequent functions, which are potentially more complex.
For example, filters allow the NAT to only select the specific state relevant for the currently inspected packet instead of retrieving a large information base for all active translation rules.

2. *Mapping Functions* are applied to transform the client specific context to the base context and vice versa. Additionally, these functions can be used to transform serialized mechanism objects within a request to the base context. This allows for minimal modifications on the client side as all mappings to the context definition are done within the context management system. In our example, the firewall as well as the SDN controller can continue to store context information using their internal state representation. In case of failure, the SDN controller is able to request additional rules from STEAN that are generated from the firewall state using mapping functions. The controller does not need to understand the state representation of the firewall but is able to use the additional information provided without adaptations.

3. *Aggregation Functions* allow for sub-context re-use. They enable the context management to combine two or more existing contexts to a single new context. Clients can register complex queries that contain information from multiple annotations within the storage system. The aggregation functions combine the results from multiple annotations and return a single annotation set to the client. Aggregations can thus be compared to the JOIN operation or views in traditional databases. After registration, clients can use complex queries for data retrieval in the same way they do for standard annotations. For example, the SDN rules generated from the firewall state (as described above) can be aggregated with the SDN rules stored by the controller to create a unified rule set that can be directly installed on the switch.

4. *Modifier Functions* are called on (filtered) data items retrieved from the storage. The functions can change the actual data within the item, alter the metadata attached to the entry, or modify custom metadata the client contributed. In our example, a modifier function can be used to add additional information from which context the SDN rules are generated. This informs the controller about the rules required for the network operation as well as the rules installed to reduce the load on the firewall.

Transformation of partial context, i.e., context information not providing the complete state required by a client, is explicitly supported. Partial context can occur when a new client is connected and other clients only gathered parts of the required state. When partial

context is available, the client can retrieve the stored information using transformation functions to convert the context but the client also has to gather the missing information using mechanism specific procedures. Then, the additional context can be contributed to the context management system and, hence, made available to other clients. For instance, one routing protocol might only be able to contribute one-hop neighbors to the context storage while another protocol also requires all two-hop neighbors of a node. When switching mechanisms, the latter can retrieve the list of one-hop neighbors from context management and start the discovery of two-hop neighbors based on this information.

STEAN-side transformations allow us to make use of a shared cache between clients when they connect in parallel and query the same context information. This cache reduces the load on the system and thus decreases the response time for subsequent requests. Additionally, we are able to reduce the communication overhead between STEAN and the clients when filtered or aggregated context information is requested as only the needed set of context information is returned to the client. The firewalls in our example use the same state representation and thus share a common cache. This results in faster access times when packets matched by the same rule are processed on either instance.

Additionally, functions executed by STEAN allow for re-use of transformations across several client implementations. It is possible to call other transformations from within a function and leverage the already implemented features without knowing the exact implementation. It also allows us to provide a library with common functionality that is implemented and loaded like any other transformation function and that can be used from various clients.

4.1.3.2 *Features*

STEAN allows clients to specify the transformation functions between their specific context and the base context upon connection. Those functions are then called each time a client reads or writes data, and the base context is automatically mapped to the client specific context and vice versa.

The mapping does not need to be a static function but can be adaptive to runtime configuration changes. This allows the client to dynamically adapt its context to the current environment without the need for redefining annotations or exchanging the transformation function. In our example above, the IDS can dynamically adapt the information retrieved from STEAN when a suspicious flow is detected and extend the number of evaluated flow properties without reconfiguration. This allows to faster detect attackers by looking for flow context stored in STEAN—that is contributed by the firewalls—once a suspicious flow is identified.

Transformation functions are designed to be modular and composable: functions can call other functions to create complex transformations with minimal effort. Additionally, transformation functions query external systems to retrieve additional information. For example, the transformation between an IP address and a MAC address requires to issue an ARP request on the local network.

4.1.3.3 *Limitations*

Transformation functions are mainly limited by their complexity and the resulting loss in performance. The complexity of a transformation not only depends on the function itself but also on the design of the base context. If the base context efficiently supports

the envisioned clients and thus the needed transformations, the overhead can be kept minimal and the performance loss is mostly negligible.

Furthermore, the client developer has to manually define the required transformation functions. Currently, there is no automatic system that generates transformation functions from either existing implicit context representations within the client (data structures, object relations), or from an explicit description of the client specific context (annotations, models).

Naturally, state transformations are further bound by the available state. That is, state can only be transformed but not inferred. For example, transforming state from a routing protocol maintaining a one-hop neighborhood to a mechanism that requires information about the two-hop neighborhood is only partially possible, as the missing state needs to be inferred by the mechanism itself.

4.1.3.4 *Designing Transformation Functions*

When designing and implementing new transformations, it is important to keep the computational overhead as low as possible since all information stored in and retrieved from STEAN potentially passes the functions. Moreover, it is necessary to evaluate the cost of using transformations against the cost of locally retrieving or calculating the information within the client without accessing the context management system. In some cases, it might be more efficient to (re-)generate the context in the client rather than extracting the needed context from STEAN using a complex transformation function. This is especially true for information with a short lifetime which requires regular updates that prevent efficient caching of transformation results.

As the complexity of the transformation functions depends on the design of the base context, a close interaction while building the base context and the transformation functions reduces the computational overhead. This includes that transformations should target a small scope of the overall context and apply filter functions as early and as restrictive as possible. Restrictive filtering limits the number of data items processed by other, potentially more complex, functions to a minimum, thus improving the response time of STEAN. This also contemplates that functions exit as early as possible: if the NAT in our example requests a single state item, the filter function must be terminated after this item is found.

During a lookup operation, transformation functions should be called in a specific order: 1. filter functions reduce the amount of data retrieved from storage, 2. mapping functions translate the base context to the client-specific context, and 3. aggregate functions then unify different data items to provide a single context to the client.

While technically feasible, transformation functions should not fetch information from external sources unless this information is a direct transformation of stored context. Additional functionality should be placed within the client as it is a feature of the implemented mechanism rather than a necessity of sharing context. In general, transformation functions should not generate new state but work on the existing context stored by the clients.

Additionally, in order to support concurrent access, all transformation functions must not directly alter the stored data but only transform the information received from the storage subsystem.

4.1.4 *System Design*

STEAN is designed as a node-local system that manages all context information of the connected mechanisms. A node is not limited to a single (physical) system but can be any network entity with a well-defined purpose. This can be instances of a virtualized NF that form a cluster of IDSes as discussed in the use case "Migrating Network Functions", or a single wireless device that is forwarding traffic in a WMN as shown in our use case "Switching Routing Protocols". A certain number of mechanisms with a well defined functionality or purpose thus form a node.

The design is centered around the transformation functions as the enabler for a generalized context management system.

4.1.4.1 *Components*

STEAN consists of five core components which are assigned specific tasks within our architecture and can be exchanged with other implementations. Figure 4.2 gives an overview of the components and their interaction.

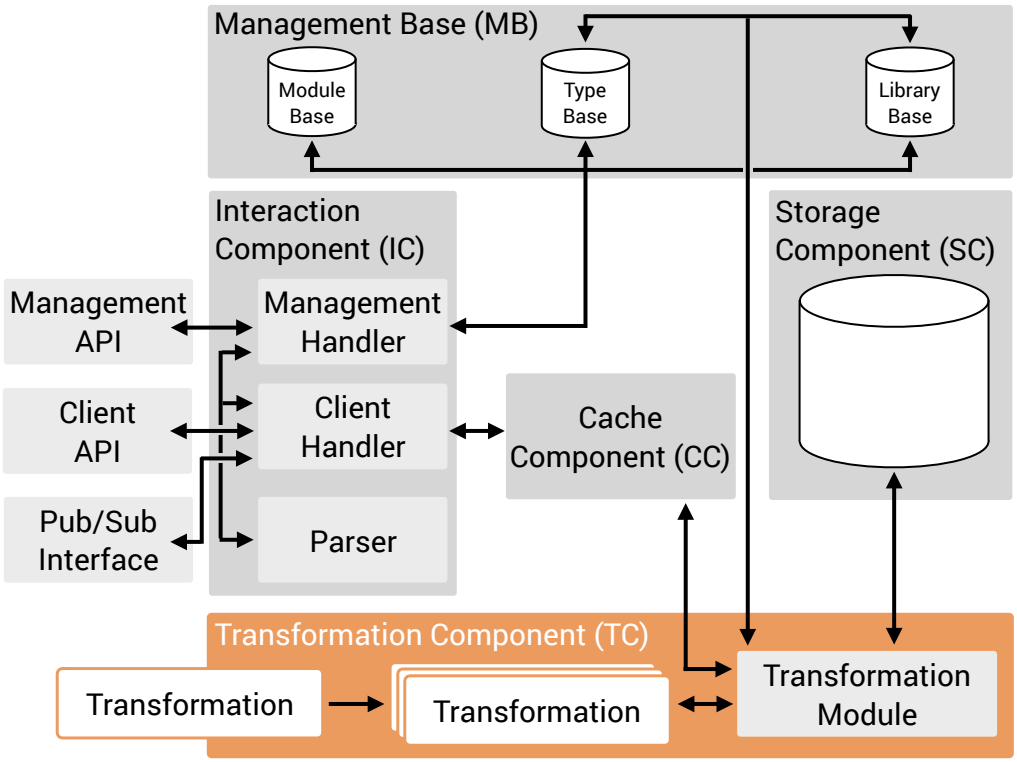

Figure 4.2: Architectural overview of STEAN. The arrows show the interaction between components. The modular mechanisms presented in Section 4.2 connect to STEAN using the Client API and the Pub/Sub Interface.

INTERACTION COMPONENT The Interaction Component (IC) is responsible for handling incoming commands. These commands can be either context requests from a client, or updates to the state of STEAN itself such as adding new transformation functions or registering additional annotations.

The IC reads a stream from a socket and dispatches the request to the responsible handler which invokes the parser and executes the corresponding backend functions. Before dispatching the command, the IC checks the command against the metadata stored in the Management Base. It retrieves information on registered annotations and available transformation functions, and returns an error if unknown requests are included in the query.

STORAGE COMPONENT The Storage Component (SC) holds the actual data that the clients add and query. Data within the SC is grouped by annotations and organized in several sets. These sets are used for efficient data retrieval since only those sets using the requested annotations have to be searched. In addition, metadata is attached to each data item. This metadata can either be provided by the client, by transformation functions that are called during the storage process, or by STEAN itself. There are two mandatory attributes that have to be assigned to each entry. The *timestamp* holds the date and time the entry was added, and the *mID* stores the identifier of the mechanism that added the entry.

TRANSFORMATION COMPONENT The Transformation Component (TC) implements transformation functions as described in Section 4.1.3. The TC is invoked on every query and connects to the SC. The TC either transforms the data retrieved by the SC to match the context of the requesting client (lookup request) or transforms the inserted data to the base context (add or modify request).

STEAN is designed in such a way that all requests have to pass the TC. However, it is possible to specify the identity function as a transformation and thus to have only a minimal overhead when no data transformation is needed. This is especially important for mechanisms that are designed to use (parts of) the base context as their local context as well.

Additionally, we provide a method to register *trusted functions* that are allowed to access data marked as *private*. These functions guarantee that they do not leak any sensitive information and can, for example, be used to cryptographically sign data with a private key stored in STEAN.

CACHE COMPONENT To be better suitable for performance critical mechanisms, STEAN makes intensive use of caching. The cache allows to reduce the retrieval costs for context lookups which is relevant for performance critical NFs (e.g., functions performing per-packet lookups at line rate).

The Cache Component (CC) is placed between IC and TC, and thus holds context information where the client specific transformation functions are already applied. The placement keeps the computational overhead of applying transformation functions as low as possible but leads to a minimal re-use of cached results across clients.

We opted against a shared cache placed between TC and IC but for a cache holding an individual set of results for each client. The diversity of clients would not allow for a wide re-use of cached entries as each client specifies its own context. A shared cache instead extends the number of entries per cache set and thus leads to a higher retrieval time.

Additionally, we decided not to place a shared cache between SC and TC. This placement would allow for a higher re-use of cached information but the gains are much lower since transformations have to be applied to each returned result.

We chose a Least Recently Used (LRU) caching policy as we assume that clients will constantly request specific annotations during normal operations while other information is only requested either during startup or at long intervals. For example, the rules filtering HTTP traffic implemented at a corporate firewall will be called regularly as employers are using their webbrowser while the rules for filtering mail traffic are called infrequently due to the fact that corporations use internal mail servers.

However, our design is open to extensions and allows for the implementation of a different caching strategy per client.

The client can invalidate its own cache and the Management API supports the invalidation of the caches of all clients. This is especially important when transformation functions are replaced and thus the caches have to be rebuilt.

MANAGEMENT BASE The complete metadata and the state of STEAN itself is represented in the Management Base (MB). The *Type Base* within the MB stores information about known annotations and possible attributes, while the *Library Base* manages the transformation functions available. The *Module Base* subcomponent holds a list of clients, and their registered annotations and transformation functions. The MB does not contain any client-supplied context, but is responsible for the state created by STEAN itself.

4.1.5 *Communication and Interaction*

STEAN provides two interfaces for outside communication. The Client API is used by the accessing mechanisms to store and retrieve context, and the Management API is used to control the behavior of STEAN itself. While the first interface is openly available to all clients, the second interface is protected to prevent unauthorized reconfiguration of STEAN.

4.1.5.1 *Client API*

STEAN supports multiple annotation sets that are registered by clients. Upon connection, each client has to register and provide the annotation (sub-)set it will use, and specify the transformation functions to convert the client-specific context to the base context and vice versa.

After successful registration, the client can access the specified annotations and transformation rules while access to other annotations or transformations is denied. This initial registration forces each client to completely model its environment and describe its specific context compared to the base context before access is granted. Changes to the set of annotations or transformation functions require a full re-connect of the client.

While the above restrictions appear unnecessary at a first glance, they provide the following advantages:

1. The set of context variables of a client is typically fixed and does not change during runtime. Therefore, sharing the context definition during registration is not a limitation but allows STEAN to accept shorter requests and to issue compact answers

that are directly mapped to the client context. This mapping highly reduces latency in communication with STEAN which is critical for networking mechanisms.

2. Providing annotations and required transformation functions during runtime would result in possible data duplication in the cache as the service might not be able to detect identical subsequent requests. Providing the context definition of the client during registration allows for a more effective caching infrastructure.

3. The service allows mechanisms to mark data as private and then only allows access to this data via trusted transformation functions. A registration of annotations that mark private data items simplifies the establishment of a trust relation between the mechanism and STEAN as the exact methods of retrieval are defined beforehand.

The mechanisms can use *add*, *remove*, *modify* and *lookup* calls to insert, remove, alter or retrieve data stored in STEAN. While the first three calls change the stored context and are directly passed to the TC and the SC, the CC can answer the last command if valid data is available. This circumvents the execution of transformation functions and storage lookups and thus offers lower latency than the other calls. Additionally, these requests are parallelized using multiple threads to allow several mechanism lookups at the same time.

STEAN also offers a publish-subscribe interface that notifies connected mechanisms when changes to subscribed annotations occur, i.e., it can notify connected clients such as monitoring systems when the stored context is altered. However, the publish-subscribe interface only sends out change notifications, and leaves the update of local context or any other appropriate actions to the subscribers. Thus, the subscriber has to react to the notifications and actively retrieve the altered information from STEAN.

4.1.5.2 *Management API*

The management interface provides methods to alter the base context of the service, add and remove annotations, and register new transformation functions. Access limitations on the interface prevent clients from registering arbitrary annotations or transformation functions that have no value to other clients (as they are unknown), or even compromise the service itself as malicious functions might leak sensitive data.

4.1.6 *STEAN Implementation*

The STEAN prototype is implemented as a stand-alone application written in C++. It runs as an independent service on any Unix-like OS. We successfully tested the functionality on Linux, FreeBSD and Apple's macOS.

Our implementation heavily relies on the features provided by both the programming language—especially in terms of low level access and pointer based operations—and the OS and programming environment (e.g., support for dynamic libraries, threading mechanisms). These features allow us to provide a fast response time with minimal latency without putting significant additional load on the node running STEAN.

4.1.6.1 *Storage System*

The storage system is implemented on top of a eXtensible Markup Language (XML) database using RapidXML [35], and the items in the database are accessible via a management plane. The management plane is implemented as a map of pointers that

allows for direct access to the requested annotation and handles the lifetime of each data item. All information is managed in memory during runtime. Hence, pointers can be used within the data structure which significantly speeds up the data retrieval.

The database consists of several sets—one per available annotation. Each data item can currently only be tagged with one annotation and is thus associated to exactly one set. To remove this limitation, the management plane provides additional indices that allow for direct access across annotation sets. These indices can be seen as virtual annotations and can be accessed in the same way. Figure 4.3 shows an overview of how data can be accessed within the store.

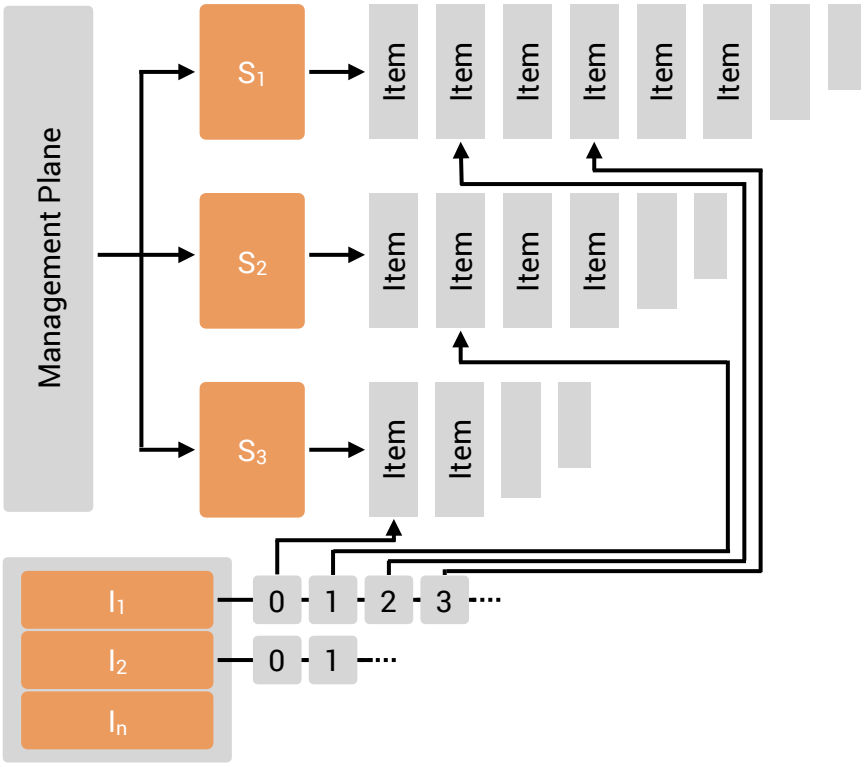

Figure 4.3: Overview of the data access within the SC: Items can be accessed via annotation sets (S) to filter by their metadata. They can also be filtered by custom indices (I), which contain pointers to items from (possibly) different annotation sets. The figure shows a storage with three sets (S_1 to S_3) and two indices (I_1 and I_2). In general, there is no limitation for the number of annotation sets, items per set and custom indices.

We also implemented a custom memory pool for the storage backend. In contrast to the implementation provided by RapidXML—that only supports the deallocation of the complete memory—our pool allows us to release specific areas of memory during runtime. This is especially important as the STEAN storage system is designed as a long living document which is continuously updated and thus requires a highly dynamic handling of memory.

STEAN also supports a *snapshot* feature that can be used to create a persistent copy of the stored information and the current state of the service. The snapshots, however, do not contain the shared libraries registered but assume that the libraries are available at the same location. To ensure consistency within the snapshot, all ongoing operations are

finished before the snapshot is taken, and all open requests are delayed until snapshot creation is complete.

The created snapshots can either be directly loaded during initialization of a STEAN instance or imported into a running instance using the Management API. This allows us to setup new instances with a known and pre-published base context to speed up the mechanism initialization phase, and replace existing context information during runtime if the operational conditions of the network change.

4.1.6.2 *Function Libraries*

Transformations are implemented as Unix Shared Objects (SOs) and have to be loaded via the Management API. This enables us to add functions on demand without shutting down or even recompiling STEAN.

After registering the library system wide, each client needs to register the used transformation functions together with the base context annotation and the mapping annotation within its client context. This ensures that STEAN calls the correct transformation function when an annotation is requested without the need to specify the function on each request, and prevents inconsistent mappings between requests from the same client. Additionally, it keeps the size of request messages low and thus increases the response time of STEAN.

The TC within STEAN offers full support of the C programming language together with all libraries available on the host. We have defined four method signatures—presented in Listing 4.1—each function has to comply to in oder to be registered as a transformation within STEAN. The signatures corespond to the transformation types defined in Section 4.1.3 and include the specific requirements of each type. For example, the mapping functions accept a simple data item that should be transformed to the target context while the aggregate functions require a more complex set of information such as a `Vector` of storage nodes that should be combined into the single annotation.

```
// Filter Functions
extern "C" pair<bool,int> name(xml_node<>* n,
                               map<string,string>* params)

// Mapping Functions
extern "C" pair<bool,string> name(string s,
                              map<string,string>* params)

// Modifier Functions
extern "C" bool name(xml_node<>* item, map<string,string>* params)

// Aggregate Functions
extern "C" void name(vector< vector< xml_node<>* > > sources,
                     vector<string>* destination)
```

Listing 4.1: Format of shared library functions

In general, a transformation function takes the XML structure returned by the SC along with a map of additional parameters—such as additional selectors, filter settings or operators to be applied—as an argument and returns a boolean that indicates the success

or failure of the applied transformation. When successful, the result of the transformation is returned as XML structure as well.

4.1.6.3 *Client Communication*

STEAN runs as a single-instance service on each node and communicates with clients either via Unix domain sockets or via TCP sockets. Each client is assigned a separate socket upon connection and communicates with our service using XML based commands.

The data received on each socket is handled as a stream of request and responses. The stream is split at the closing tag of the XML element by the IC. Malformed messages are either ignored or answered with an error message, depending on how much of the message can be interpreted. All messages are then classified to determine the component responsible for handling the request. Messages that update the MB are directly passed to the component while requests to the data storage are forwarded to the TC.

The response is passed back to the client handler that uses a message builder to create valid XML and return the answer to the client.

STEAN supports concurrent access of multiple clients. The service uses a synchronized queue to manage incoming requests after classification and to forward lookup operations to the addressed component. We use a scheduler to continuously read from this FIFO queue and forward the requests to the responsible component.

Read operations are immediately forwarded by the scheduler and processed in parallel to reduce the latency of a request. This requires the transformation functions to not alter any data items in the storage backend during lookup operations (including the metadata) as this would contradict the fully parallel operation mode.

When the service receives a request to alter the stored data, the execution is blocked until all currently running operations are finished and the storage is in a consistent state. The operation is then executed and the scheduler blocks all other subsequent requests until the alter operation is finished. Currently, our implementation blocks all operation regardless of which part of the storage they require to access.

4.1.7 *Client Implementation*

We designed and implemented a client library that provides convenient access to STEAN without the need for the client developer to handle the connection management and the XML message building and parsing. Additionally, we provide fully functional client implementations for two mechanism classes—that are in the focus of the use cases presented in Section 3.1—along with the required transformation functions to support the seamless transition between those mechanisms.

Client side caching potentially brings additional performance benefits to mechanisms connected to STEAN, depending on the access and update behavior. It can dramatically reduce the latency when continuously retrieving the same information but is rendered useless if other connected clients are permanently updating the requested data. While we provide a library to connect client mechanisms to STEAN, we opted against including a generic cache solution. This allows us to focus on the core functionality and enables mechanism designer to create a caching solution that specifically targets the requirements and singularities of their mechanism.

4.1.7.1 *Network Functions*

The STEAN support for NFs is built on top of the OpenNF [28] modifications that allow to migrate NF state between different instances. Instead of migrating the state via the controller, we directly share context information between the NF instances using STEAN and thus circumvent the bottleneck of passing all context information through the already loaded orchestration instance.

We have chosen the PRADS asset monitor [24] as an example to show the feasibility of our approach. PRADS is a passive network monitor that allows operators to map the services running in a network and detect changes in real time. It uses TCP and UDP fingerprinting to identify OSes and service applications. PRADS also keeps an internal state table to identify flows in the network and provide information on the services offered and used by the networked systems.

We modified PRADS to be a STEAN client while still supporting the OpenNF controller messages to initiate the migration of flows between instances. The PRADS instances share the complete internal state of all observed flows using STEAN. The per-flow state generated by PRADS consists of a unique identifier per flow, the protocol 5-tuple and the IP protocol version along with timestamps for the first and last seen packets, the number of packets observed for the flow, as well as the total size of transmitted data for each direction. PRADS also includes the hardware protocol and any TCP flags observed into the flow state along with a list of identified assets for source and destination.

We specifically target the use case of migrating NFs with our implementation, and thus we are only sharing information between instances running the same implementation. The only transformation functions required are filters to select specific flow entries based on the unique flow identifier assigned by PRADS. Therefore, also the base context is held simply by mapping the identifying hash of a flow to the item key and the serialized data object that comprises the flow properties to a `String` element as shown in Figure 4.4.

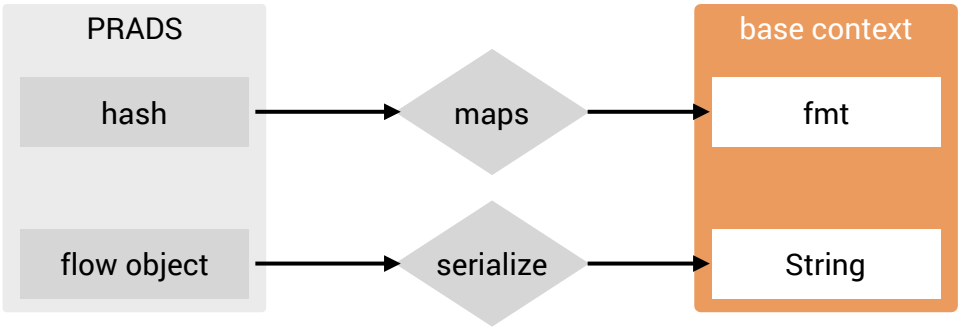

Figure 4.4: Mapping of the internal PRADS state elements to the base context.

4.1.7.2 *Modular Routing Protocols*

We modified implementations of the Ad hoc On-Demand Distance Vector Routing Protocol (AODV) as well as the Optimized Link State Routing Protocol (OLSR) to support the use case of switching routing protocols as introduced in Section 3.1.4. The mechanisms are implemented using Click, and we extended the state handling elements to connect to STEAN. Each mechanism uses a special Click element that is responsible for specifying the mechanism context, and registering annotations and transformation

functions during system startup. This element also handles the communication with STEAN during mechanism operation.

The separation of the mechanism functionality and the context handling allows us to keep the modification overhead in the existing implementations low. It leads to a strict separation of concerns: the STEAN client can be exchanged with another implementation without changing any other code. This allows us to evaluate clients with different data access and caching strategies.

Our implementation makes heavy use of client side caching and thus serves as an example how a client side cache that takes the quirks of the implemented mechanisms into account, can be realized. We use the publish-subscribe interface provided by STEAN to register for annotations used by the mechanism implementation and invalidate the client-side cache entries when the information stored in the context management system changes.

Accompanying the implementation changes, we designed a base context that closely matches the requirements of the routing protocols. Specifically, we share the list of one- and two-hop neighbors as well as the list of multipoint relays and the routing tables. The mechanisms do not hold any local context but solely access information stored in STEAN. Figure 4.5 shows the relation between the information used within the routing protocols and the base conext used by STEAN.

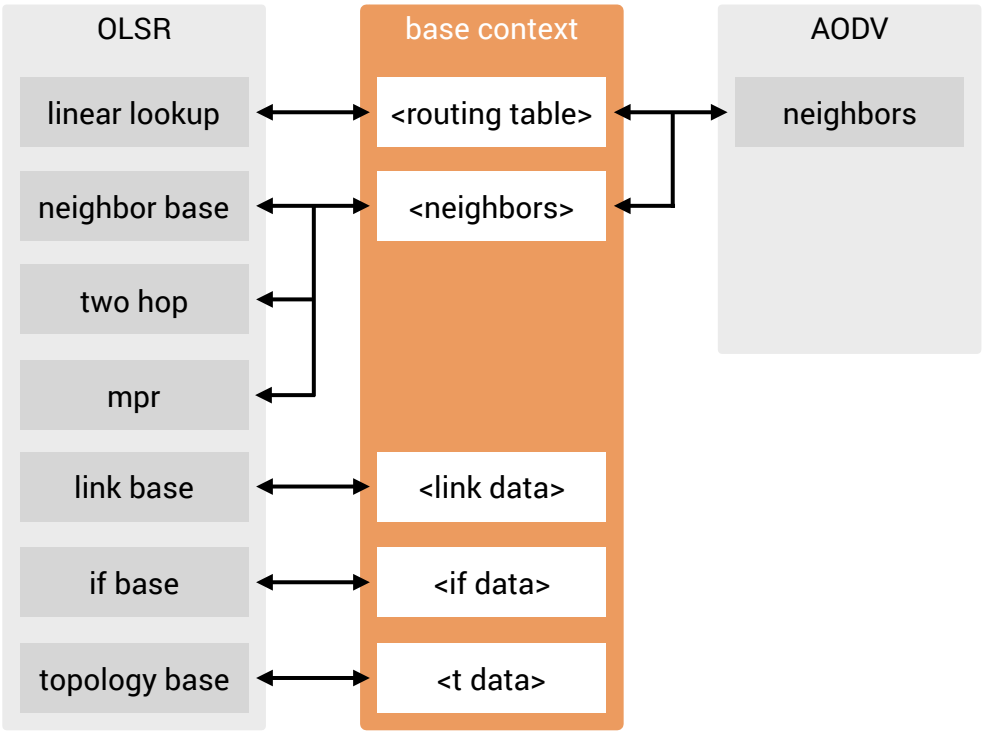

Figure 4.5: Relation between the mechanisms specific context representation of OLSR, AODV and the base context.

We have implemented transformation functions that convert the base context to the specific context of each mechanism and vice versa. These functions include an aggregate function to join the information into one result as needed by our AODV implementation

as well as filter functions that allow us to select entries from the shared context and thus only extract the information needed to forward the currently processed packet.

4.1.8 *System Analysis*

In this section, we present a general analysis of our design and implementation. Our goal is to understand 1. the performance of basic STEAN operations, and 2. the scope of changes required to adapt existing mechanisms to our context management system.

A more extensive evaluation that shows the performance of STEAN in multiple use cases is performed in Chapter 5.

4.1.9 *Performance*

We analyze the overall system performance of STEAN by evaluating the insert and retrieve time of context information. We use a simple client that inserts and reads IPv4 addresses that are either represented as a `String` with dots separating the octets or each octet represented as a `Integer` value. Additionally, transformation functions are available in STEAN to convert between these two formats.

The evaluation is conducted on a single machine with a Quad-core Intel Xeon CPU and 16 GB of memory. All caches are disabled to show the raw performance of the transformation engine and the context storage.

Figure 4.6 shows the time per insert for inserting 1000 (a, d), 10.000 (b, e) and 100.000 (c, f) unique IPv4 addresses both with and without applying the transformation function.

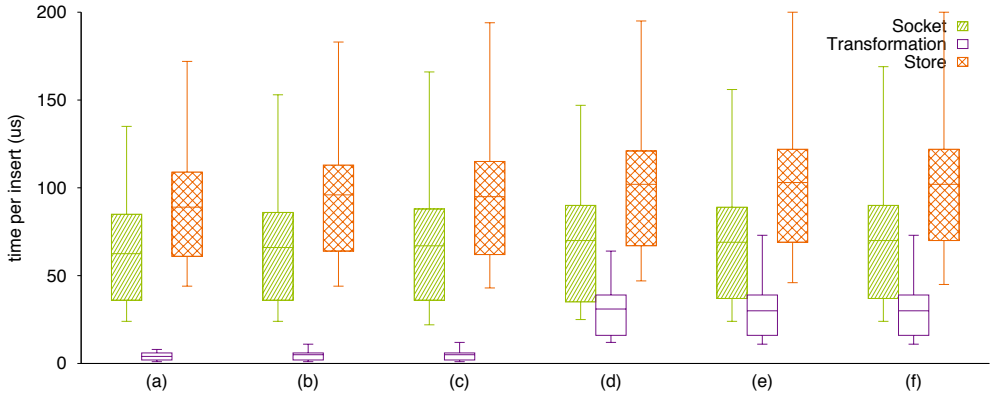

Figure 4.6: Time per insert without calling a transformation function (a–c), and with calling a transformation function to convert the representation (d–f).

Our results show that writing to STEAN takes constant time regardless of the number of entries already stored. Saving one context entry takes about 140 µs when no transformation function is employed and around 180 µs when the simple function described above is used to convert the representation.

Additionally, we see that at least 1/3 of the request completion time is spent on socket communication.

The time shown for transformations, even when no function is executed, is due to the overhead passing all requests through the transformation engine and not interfacing

with the storage directly. While we focus on a single client in Figure 4.6, we remark that additional clients have a negligible performance impact and only increase the variability of the insert time (not shown).

Figure 4.7 depicts the results for reading one out of the 1000 (a, d), 10.000 (b, e) and 100.000 (c, f) addresses inserted before. The address is selected by applying a filter function for a random but fixed address per operation.

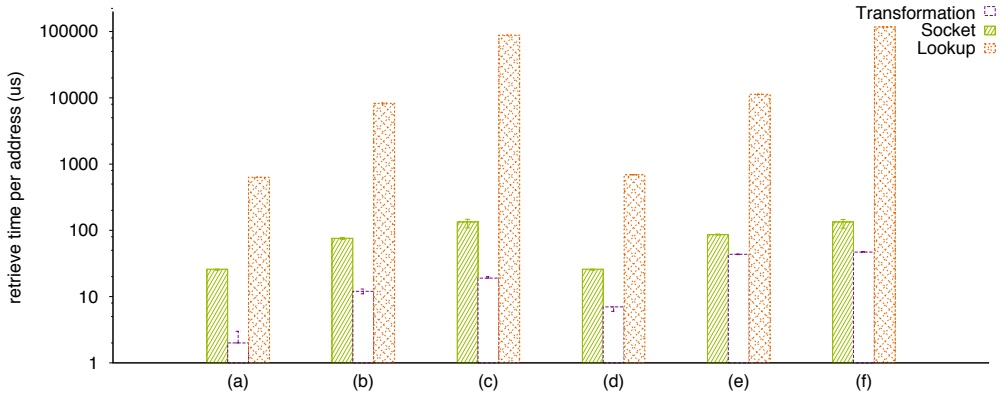

Figure 4.7: Time to retrieve an address without calling a transformation function (a–c), and with calling a transformation function (d–f). The boxes represent the median and the error bars show the first and third quartile.

The experiments show that the time for retrieving context is linear to the number of entries stored in STEAN. This is due to the current implementation of the storage component that is iterating over all entries for an annotation until a match is found.

This behavior is also represented in the timings for the transformation engine as they include the time for applying the filter function. Each item is passed through the filter to check for a match and thus the transformation time also increases with the number of entries. Concurrent lookup requests do not influence the performance of STEAN as read operations are executed in parallel.

Our initial system analysis demonstrates, that STEAN is able to handle large amounts of context information with only minimal overhead, and is thus suited even for highly loaded networks. In Chapter 5, we strengthen this conclusion by integrating STEAN into existing network architectures, and applying different work loads from our use cases.

4.1.10 *Implementation Overhead*

To quantify the modification required to support STEAN using the mechanisms presented in Section 4.1.7, we counted the Lines of Code (LoC) that were added or changed in each implementation and present our findings in Table 4.1.

The results show that systems designed to share context information only require minimal changes to support STEAN. While the number of LoC for AODV and OLSR suggests rather dramatic changes, the actual implementation overhead was minimal since only a few functions needed to be changed.

However, as these functions were largely scattered over the code, they increased the overall LoC count. Nevertheless, most of these changes can be generated automatically using refactoring tools.

IMPLEMENTATION	LOC ADDED/CHANGED	CHANGE IN CODE
AODV	542	21.4 %
OLSR	1289	49.9 %
Common Click Code	1243	n/a
PRADS w. OpenNF	144	0.7 %
STEAN shared library	972	n/a

Table 4.1: Additional or changed code to implement STEAN support.

4.1.11 *Summary*

We presented STEAN—a Storage and Transformation Engine for Advanced Networking context—that enables us to not only share state between instances of the exact same implementation but to extend the sharing of network context beyond these boundaries.

STEAN supports the decoupling of functionality and context and serves as the state plane that is responsible for gathering the state information from mechanisms deployed in a network segment, and enriches the gathered information to provide the network context to the connected mechanisms.

We introduced the concept of transformations to bridge the gap between existing mechanisms that do not share a common context by adding a translation layer that mediates between the mechanism specific state representation and the base context used by STEAN. This additional layer allows us to leverage the advantages and features of a context management system with only minimal changes to existing implementations. It also enables a faster development of new mechanisms as it reduces the overhead of handling state information locally by providing a well defined API along with supporting tranformation functions. The developers can focus on the functionality and leave the context handling to STEAN.

Our component based architecture is centered around transformation functions as the core feature and enables us to connect arbitrary mechanisms from WMN routing protocols over BGP implementations to virtualized NFs with a minimal performance penalty. This penalty is further reduced by introducing a specialized storage backend that is based on a in-memory XML database and a customized memory pool that efficiently supports high frequency updates by using pointer operations on the stored information.

We also implemented a message cache that uses per client results with tranformations already applied to efficiently support reoccurring requests from the same mechanism. This avoids the multiple execution of potentially costly transformation functions while still providing a performance gain for the connected clients even if they are not sharing the cached results. We support multiple caching strategies on a per client basis to provide an optimized solutions to all connected mechanisms and to further reduce the latency overhead. An evaluation of the gains provided by the cache is presented in Section 5.2.

Additionally, we offer client implementations for two distinct use cases, namely “Switching Routing Protocols” presented in Section 3.1.4 and “Migrating Network Functions” discussed in Section 3.1.1, based on the Click Modular Router and OpenNF respectively. The client implementations show how STEAN can be incorporated into a diverse set of mechanisms that depend on different frameworks without the need to adapt the inner

workings of the mechanism but replacing the internal state handling with a set of library calls to STEAN.

4.2 a blueprint for switching between mechanisms

In this section, we present the blueprint of an architecture for transition enabled mechanisms, and introduce a framework to execute such a mechanism switch during runtime. To this end, we deviate from monolithic mechanisms and follow a modular approach, thus further increasing the flexibility of our solution.

We focus on the use case of switching WMN routing protocols discussed in Section 3.1.4 as it is one of the most challenging examples in providing a switching architecture both in terms of mechanism design as well as coordination during the switch.

Currently, many configuration parameters and settings of a WMN are chosen at deployment time. This includes the choice of a routing protocol and the associated protocol parameters such as link weights, timeouts etc. that vary depending on the protocol type and mode of operation. The choice depends on the application scenario and the existing boundary conditions—current mechanisms for WMNs are typically tailored to a narrow scenario set—thus yielding the desired performance.

These conditions are subject to change over time. A once stable and reliable network can suddenly get unreliable due to external factors such as obstacles that obstruct wireless links or interference caused by other radio devices. Moreover, in WMNs the topology can change completely over time as mobile nodes move within the network or as the network organically grows and new applications are deployed. These changes in the environment and the network's structure as well as its usage may require changes in the routing protocol during runtime to maintain the desired network performance. Yet, current WMN deployments do not support dynamic transitions between different routing protocols.

While nodes in an infrastructure based network are usually managed by a single administrative domain, the nodes in a WMNs are often controlled independently of each other. This also includes the availability of complete mechanisms and mechanism features as the implementation can be different on each participating node. Therefore, we need to ensure that a synchronous transition between the routing services is supported by all nodes.

One solution to this problem is to deploy a set of mechanisms and then select on demand the mechanism fitting the current situation best—switching between those choices during runtime. However, we then need to gather the routing information for the new mechanism and establish new routes before the network is operational again.

4.2.1 *Assumptions*

A core question is: when should one mechanism be exchanged with another one and with which one. For this purpose, we can either employ a traditional monitoring solution such as Nagios [50] or Icinga [69] that uses agents installed on each host, and connect back to a central instance.

The central service analyzes the collected data using a set of pre-defined metrics to observe the network performance. The global view enables a central monitoring to provide concise information on the current operational status of the complete network

and supply the operator with an up-to-date view of the system performance and potential bottlenecks.

However, those systems are built for monitoring networks that use a rather static topology, and are operated by a single administrative instance where a central controller orchestrates all networked devices.

WMNs are, however, built with a decentralized approach in mind. Therefore, monitoring solutions that are specifically tailored towards WMNs such as Damon [58] or Crater [60] are available. Those systems collect metrics in a distributed fashion and the monitoring agents deployed on the network nodes already process, aggregate and correlate the collected information before forwarding them to neighboring nodes or a data sink to reduce the load on the wireless channel. Monitoring services designed for WMNs are able to 1. dynamically adapt to the underlying network topology, 2. handle the characteristics of a wireless communication medium to distribute the monitoring results, and 3. take the resource limitations of mobile devices (e.g., energy) into account.

For now, we assume the existence of such a monitoring service—either centralized or distributed—that is able to support us with connectivity and performance metrics in order to support our decision on initiating a transition.

Still, the question remains which system decides to initiate the transition and instructs the nodes to change the used service and when, i.e., which system coordinates the switching. This system needs to incorporate the information collected by the monitoring service as well as additional requirements either supplied by the operator or users such as certain QoS levels, by applications running on top of the network infrastructure, or by the (expected) environmental conditions.

The decision and execution engine could again be a centralized system, orchestrating the complete network from a single instance or a distributed decision scheme such as majority voting. While a centralized system is able to incorporate monitoring information from the complete network, and collect and consider all features that are requested by the users, the view of a decentralized system can be limited due to connectivity problems or even a split in the network. Still, as WMNs are distributed systems by design, a distributed decision engine is usually better suited to those networks as a central—and thus critical—instance is explicitly avoided.

The remainder of this section is based on the assumption that a decision and coordination system exists and that the system is able to contact and control all participating nodes to orchestrate the transition.

In addition, we need to develop ways to assure that the coordination of the nodes' transition process is protected by an adequate authentication scheme when the commands are issued on an in-band connection that is also used for data traffic. Otherwise, potential attackers could flood the network with switching requests which would cause the entire network to fail in the worst case.

As ensuring the authenticity and integrity of messages is a general problem in WMN (routing) mechanisms, we omit this question for now. We assume that the communication with the other nodes is established via a secondary channel such as a very robust but low bandwidth mesh network. A low bandwidth channel is sufficient as only status information and switching commands are exchanged but no data traffic is routed over this network.

4.2.2 *System Design*

In this section, we describe our approach to modular mechanisms and present a solution for developing and deploying mechanisms that are designed with sharing of network context in mind. Our design serves not only as an example on how modular mechanism architectures can be used to share context information. It also acts as a guideline for other developers on how frameworks for modular mechanism design can be used to support context sharing.

We focus on the clear separation of functionality and context to ease the sharing of information between mechanisms by either common elements available within the framework or by employing an external context management system—such as STEAN—that is connected to each mechanism implementation. Figure 4.8 gives an overview of the design and the interaction of the components involved in our architecture.

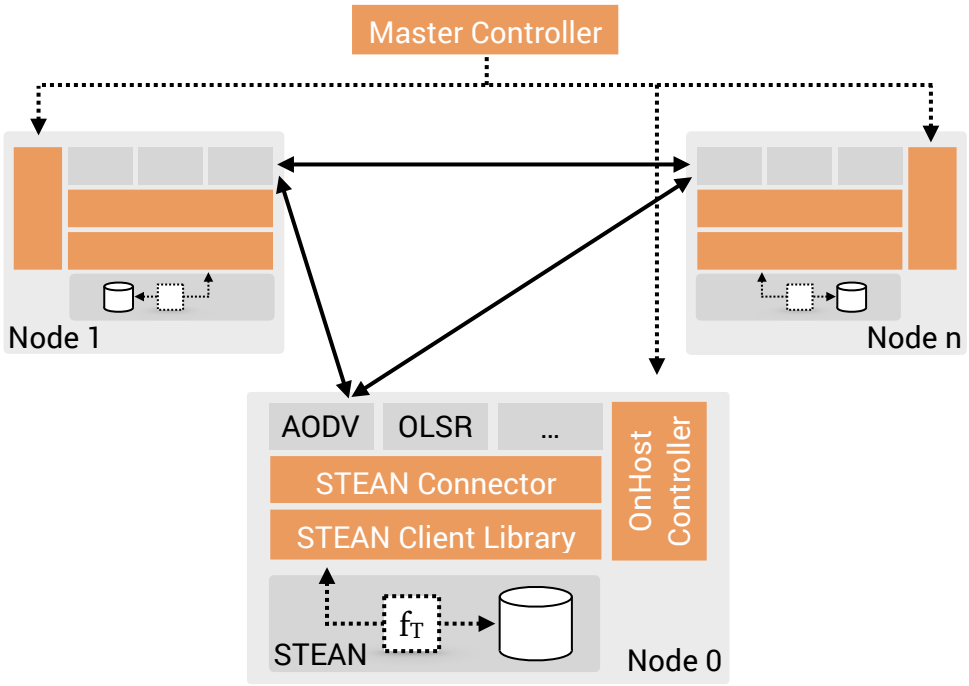

Figure 4.8: Overview of the switching system and the component interaction.

While our focus is on the design of transition aware mechanisms, we have also developed an infrastructure for the switching system that is based on the assumptions presented in Section 4.2.1. This system supports the evaluation of our approach and serves as a guideline to develop and deploy the services currently not included in this work.

4.2.2.1 *Controller Infrastructure*

Our design includes a hierarchical architecture of controller elements that are responsible to transfer and forward the instructions issued by either the operator or the orchestration system to each mechanism implementation that is actually handling the data traffic. The placement of the different controller elements and the interaction between them is shown in Figure 4.9.

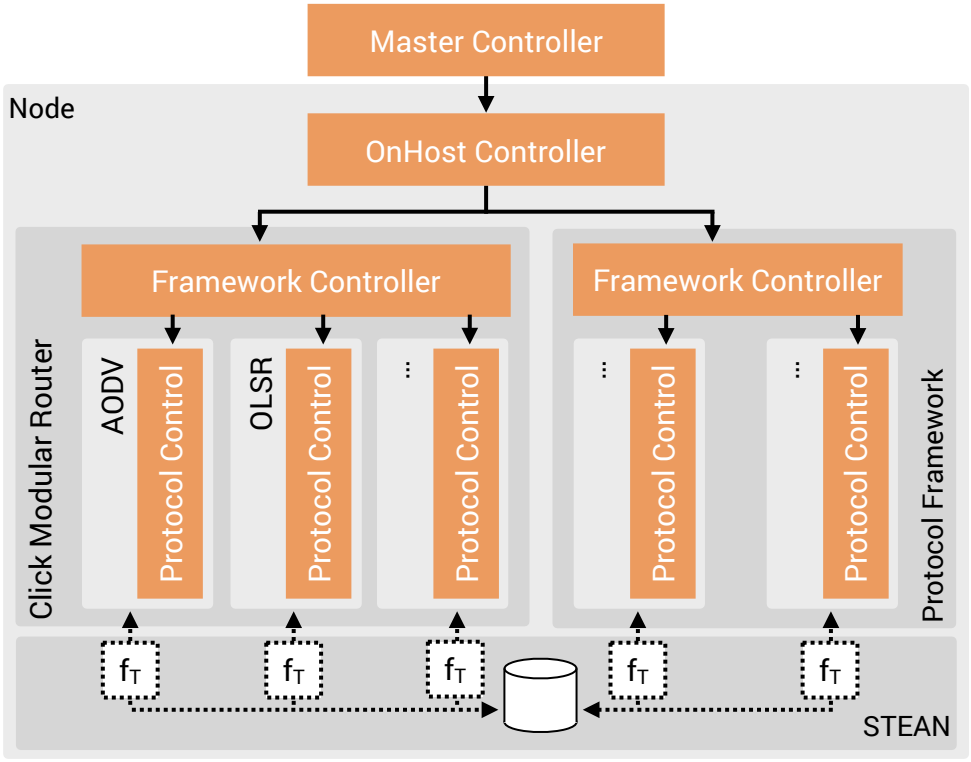

Figure 4.9: Placement and interaction of the controller infrastructure.

MASTERCONTROLLER The MasterController is the central interface to control the network and serves as a stub for the monitoring services and the connected decision engine. When a new node joins the network, it registers with the MasterController and announces the available routing protocols together with a list of optional features supported by the implementation.

The MasterController provides methods to initiate the switching process and orchestrates the switch to a new routing protocol. The controller can either reside on a separate node in the network or on a normal WMN node.

ONHOSTCONTROLLER We designed the OnHostController as a proxy between the MasterController and the mechanism implementations that translates the commands issued by the MasterController to framework specific instructions that are understood by the implementation.

The OnHostController connects back to the MasterController on startup and announces the available mechanisms. It then listens for switching instructions from the MasterController and interfaces with the routing protocols locally available, forwarding the received commands.

We chose to introduce this proxy service to stay independent from the mechanism framework used and possibly employ different frameworks on different nodes without the need to adapt the MasterController. This even allows us to provide different implementations of the same mechanism on a single node without the need to integrate them into one common mechanism framework.

FRAMEWORKCONTROLLER The FrameworkController offers a common interface for all mechanisms that are implemented in the same mechanism framework. It offers a single interface for the OnHostController to instruct the implemented mechanisms and is responsible for dispatching the commands received from the OnHostController to the mechanisms available within the framework.

It is also responsible for controlling the traffic flow within the framework, and ensures—in conjunction with additional modules—that the selected mechanism processes and forwards the data packets received by the node.

This additional layer allows us to perform complex orchestration scenarios such as dynamically adjusting timers, automatically redirecting data and control traffic within the framework to selected mechanisms, or even duplicate traffic to be received by multiple mechanisms in preparation of a transition without offering specialized interfaces or adapting the OnHostController.

MECHANISMCONTROL Each mechanism implements a MechanismControl module that serves as a single source for the definition of properties, requirements and features of the mechanism. It also announces framework specific information such as other elements, the mechanism implementation is connected to.

The MechanismControl orchestrates the other modules composing the mechanism such as enabling and disabling mechanism specific timers that would otherwise consume CPU time or even generate unwanted traffic (e.g., HELLO messages).

The element registers the mechanism with the FrameworkController on initialization and thus allows to add additional mechanisms without adapting the FrameworkController or specifically define the available mechanisms beforehand. By using the MechanismControl element, we are able to dynamically load mechanisms during runtime if supported by the framework.

4.2.2.2 *Modular Mechanisms*

The modular mechanisms handle the actual data traffic as well as manage their specific control traffic such as the neighbor announcements and the route discovery.

We encapsulate each functionality into modules at a very fine grained level and use dedicated context modules where necessary to aggregate and unify the access to context information per mechanism. This helps us keep the information gathered by the module close to the origin while still providing a unified interface to an external context management system such as STEAN, and therefore eases the transition between different routing daemons.

The transition between mechanisms can be seen as the transition between a set of modules where each module is either re-used in the new routing scheme or is responsible for handing over its information to the corresponding new module.

The context transition can be performed using a direct exchange between the involved modules or alternatively by storing/retrieving the information in/from a common context management system. While the former keeps the relevant information closer to the functionality, the latter allows for a wider sharing of context. It allows mechanisms to share their context across framework boundaries, and enables them to employ context transformations to adapt the stored information to the desired representation.

The mechanism specific operations such as exchanging routing information or forwarding traffic are executed on the data network and run independently from the control commands sent by the switching system.

Mechanism designers should keep the modules for the operation of the core mechanism features separated from the elements responsible for switching support to ensure that the network continues to operate even if the controller infrastructure is temporarily not available. However, each mechanism implementation must provide a specific MechanismControl element as described above to announce itself to the FrameworkController.

4.2.2.3 *STEAN Connector*

The STEAN Connector offers a unified interface for modular mechanisms to store and retrieve context information using the context management system introduced in Section 4.1. The element serves as an abstraction layer for the STEAN API, and allows us to design and develop mechanisms independently of the STEAN implementation. It enables us to extend or even exchange the context management system without the need to adapt all mechanisms to the new architecture.

The connector maps the internal calls of the mechanism implementation to STEAN API calls. Therefore, existing implementations do not need to be altered, and the function calls for accessing state information remain the same. This eases the transition towards sharing context information as only a small subset of the implementation needs to be adapted.

To acomplish the mapping of calls from arbitrary mechanisms, the STEAN Connector dynamically loads the concrete implementation of the mapping functions on initialization using the same mechanisms as the MechanismControl element described above. Thus, we provide developers with a single interface to integrate the mechanisms into our framework.

Additionally, the connector provides a client side cache to reduce the amount of subsequent calls to the STEAN API. The cache supports multiple strategies that can be dynamically selected by the mechanism in order to fullfil the desired caching properties.

Placing the cache close to the mechanism implementation allows us to provide a tailor-made solution without leaving caching to the mechanism developer even though each mechanism requires its own cache instance—as already described in Section 4.1.4.1—due to the diverse set of context information.

The STEAN Connector uses the library introduced in Section 4.1.7 as a convinient way to access the context information managed by STEAN. The library allows developers to use standard function calls from within the programming language of choice instead of dealing with the XML API directly, thus facilitating the development of mechanisms.

4.2.3 *Implementation*

We developed a prototype of the switching system using Click [39] as a framework for modular mechanism design. Click supplies us with a Domain Specific Language (DSL) to rapidly prototype and built not only routing protocols but also other types of mechanisms such as a NAT.

The framework provides multiple components called *elements* where each element only offers a very small but well defined functionality such as inspecting packets, cryptograph-

ically signing a payload or modifying the IP header. Multiple elements are then linked in a directed graph forming the functionality of the mechanism, and packets flow along the edges of this graph.

Click allows for two modes of operation: 1. a binary running in user space that loads a specific configuration on startup, and 2. a kernel module that uses a special filesystem to communicate with the user space and allows for dynamic reconfiguration.

However, our implementation is currently only available for the user space mode as we rely on external libraries that are not available in kernel mode such as the Botan Cryptographic Library [45]. Enabling kernel mode would require to port the library functions to separate Click elements.

In particular, Click implements the modular routing protocols, the STEAN Connector and the FrameworkController as shown in Figure 4.10. Currently, we provide implementations for AODV [55], a reactive WMN routing protocol, and OLSR [32], a proactive protocol, as well as their secured versions: the Secure Ad hoc On-Demand Distance Vector Routing Protocol (SAODV) [29] and the Secure Optimized Link State Routing Protocol (SOLSR) [2].

The mechanisms themselves are composed of multiple elements to clearly separate the routing functionality from the cryptographic operations, and enable a re-use of common elements across the secure and the non-secure versions. The mechanisms also share common functionality such as the ARP handling or the access to the physical network device.

Each mechanism implementation includes a separate MechanismControl element that announces the mechanism identifier to the FrameworkController and registers the implementation with our switching system on startup.

We also developed additional modules to support the mechanism switch within the framework. This includes a TrafficSwitch that is able to pass the data traffic to the currently active mechanism for processing and forwarding to the next hop as well as a configurable MechanismClassifier that is able to dissect the incoming control traffic based packet characteristics such as the destination port.

The separation of handling control and data traffic within the framework allows us to run two mechanisms in parallel where both mechanisms exchange control information with their neighbors but only one mechanism is forwarding data traffic.

The MasterController and the OnHostController are implemented as standard Unix daemons running in user space and communicating with the Click modules using TCP sockets. The framework offers *handlers* to control the behavior of an implemented mechanism during runtime outside of the data path used for processing packets. We use these handlers in our implementation to interact with the FrameworkController which aggregates and dispatches the commands from and to the MechanismControl elements.

The internal interaction between FrameworkController and MechanismControl is implemented using direct function calls which reduces the communication overhead and speeds up the processing of control information.

The separation of controllers has the advantage that the routing framework does not have to be adapted if new features are introduced or if one of the controllers is replaced with a more advanced or distributed version. At the same time it is also possible to replace the mechanisms with another implementation without the need to replace the controllers.

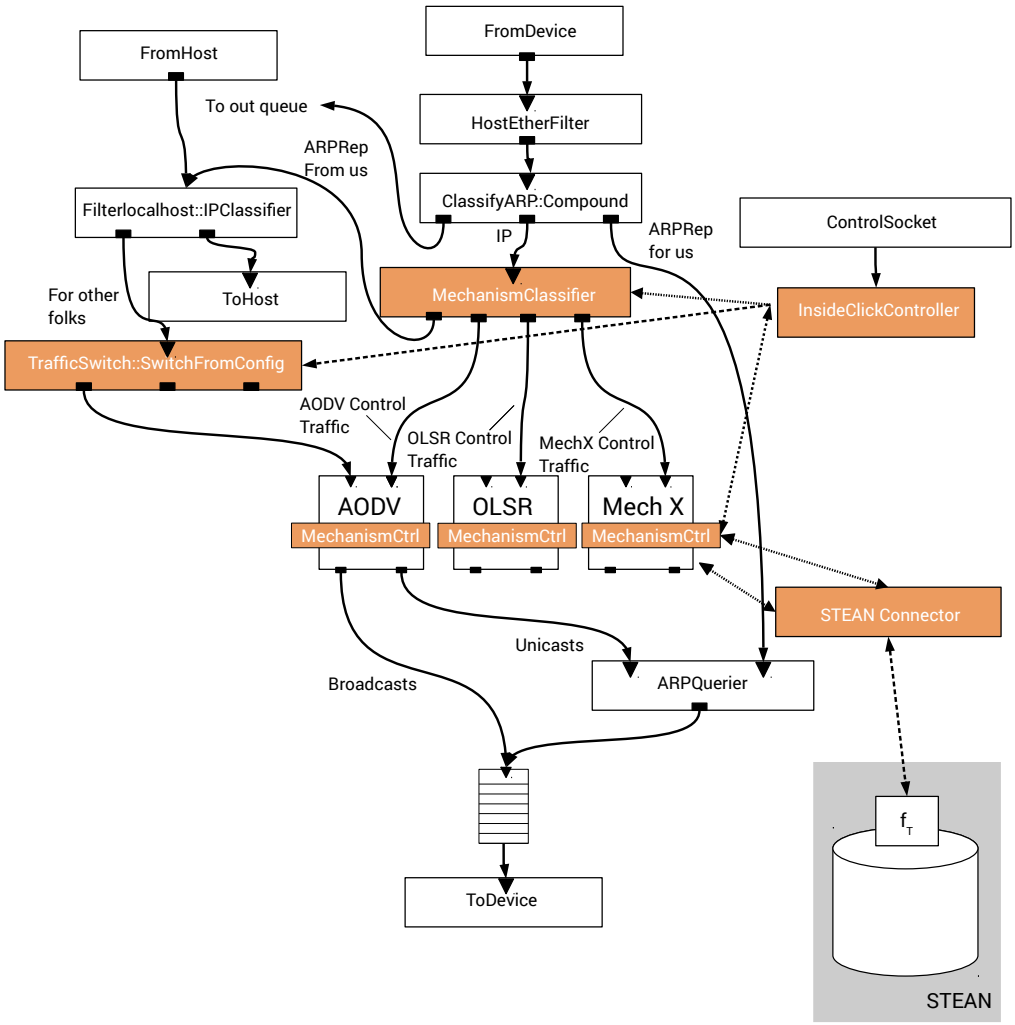

Figure 4.10: Implementation of the modular mechanism design in Click.

As Click currently only runs single threaded and in order to keep the execution of the current configuration from blocking, handlers are not waiting for results as this would also block the forwarding of data traffic. Hence, the handlers need to be polled for new or updated information. The polling is done by the OnHostController and once a change in status occurs the MasterController is notified.

The OnHostController connects back to the MasterController using a reliable out-of-band connection that provides connectivity even if the main link—handled by the Click implementation—fails. The use of a separate control channel ensures that the communication between the central MasterController and the connected nodes—running the OnHostController—is never lost even if a mechanism transition fails and some nodes run a different mechanism than the other nodes (i.e., the network is split). This is especially critical if mechanisms that are responsible for the end to end connectivity such as the routing protocol are included in the transition. While a split network situation disrupts the forwarding of data traffic between end hosts, the secondary link allows us to recover from this error by simply issuing another switch command that enables the same mechanism implementation on all nodes again. However, our implementation is also

able to use the main data link between nodes for its control information, if a secondary channel is not available but then leaves the failure recovery to the individual node.

The MasterController is implemented as a single instance component that runs only once in the complete network and acts as the orchestrator for the switching process where the mechanism information of all nodes is aggregated and offered to the network operator in a unified and concise view.

As we focus on the transition between mechanisms, we currently also do not incorporate a monitoring solution to observe the network behavior and automatically base switching decisions on this information but handle the decision to switch manually. This allows us to control the transition between mechanisms without relying on the results of a decentralized decision engine or monitoring system.

For this purpose, the MasterController has a command line interface where switching requests can be issued. Those requests are then distributed to the mesh nodes in a synchronized fashion. The interface allows us to automate the mechanism transition during an experiment and execute multiple switches in sequence without manual interaction.

4.2.4 *Summary*

In this section, we presented an architecture that allows us to switch between routing protocols in a WMN. We have defined a blueprint for transition aware mechanisms that enable a seamless switch between mechanisms during runtime by sharing context information such as the routing table.

Our approach serves as design guideline for mechanism architects to create mechanisms that are developed and built with context sharing in mind. We have based our approach on the principle of modular mechanism development and extended the architecture to natively support transitions within the complete mechanism stack.

Additionally, we have shown how existing mechanisms can be modified to support both a mechanism switch as well as context sharing and how these two principles can be incorporated into a revised implementation.

To this end, we have introduced a hierarchical set of controller components (MasterController, OnHostController, FrameworkController and MechanismControl) as well as a joint connector element that allows the sharing of context information using STEAN.

However, the presented architecture is not limited to the sharing of routing information but targets the complete set of context information on all layers of the network stack.

While we focused on WMNs in this section, the presented architecture principles are not limited to one networking technology but can be applied to other network designs or use cases, and the results can be easily transfered to both wired networks such as SDNs as well as other wireless technologies.

Our architecture enables all mechanisms to connect to the state plane of the network and benefit from the extensive sharing of context. It enables network operators to dynamically select the running mechanisms during runtime without the need for downtimes while a transition is executed.

We have implemented a prototype based on the Click framework, which enables a switch between the WMN routing protocols AODV and OLSR as well as their secure counterparts SAODV and SOLSR. For this purpose, we modified the existing implementations and added multiple elements both in the mechanism design as well as the surrounding framework.

Additionally, we have implemented the controller infrastructure to dynamically select mechanisms during runtime and disseminate the selection to all connected nodes.

An evaluation of our framework along with a discussion of the insights gained from running the system in a WMN testbed is presented in Section 5.2.

4.3 DISCUSSION

In this chapter, we presented the two major building blocks for enabling context sharing across multiple mechanisms and laid the foundation to establish a network wide state plane.

First, we presented STEAN, an implementation of a context management system that collects, aggregates and disseminates context information across all connected mechanisms. Our design enables us to store and retrieve context information from any connected mechanism without the need to adapt the internal state representation of the contributing instance to the context representation used by STEAN.

We realized the mechanism independence by introducing transformation functions that enable STEAN to construe the mechanism specific context representation from a common base context that is used within the context management system and that represents the complete set of context information usually scattered across the network segment.

STEAN acts as a network wide state plane that provides a unified interface to state information and enriches the stored data to provide the network context. It offers transition support for both legacy implementations as well as newly designed mechanisms.

Second, we introduced a structural design to enable mechanism transitions during runtime without service interruption. Our design is based on a distributed controller architecture that allows us to connect multiple mechanisms to a decision and control engine that orchestrates the switching process. We provide generic interfaces that are independent of the underlying mechanism and unify the command and control channel into a common architecture.

Additionally, we show how mechanisms need to be altered to support seamless transitions by introducing additional elements into modular mechanism frameworks, dedicated to switching support. Using distinct elements allows us to reduce the changes to the actual elements composing the mechanism functionality to a minimum and thus increase the adaptation rate of existing implementations.

We specifically do not assume a clean slate approach which ignores existing implementations and requires developers to implement major changes to their existing mechanisms. The targeting of specialized mechanisms in such an approach allows architects to create high performance implementations with virtually no additional overhead. However, it ignores the already existing and widely deployed legacy mechanisms in favor of a highly customized network architecture.

The state handling in such mechanisms is usually only driven by considerations about forwarding performance of single mechanisms, but ignores the fact that cooperating functions can lead to a better overall network performance even if individual mechanisms operate at a lower forwarding rate.

The sharing of context information across multiple implementations and even among mechanisms with different functionality is not in focus of existing state of the art state handling systems. Instead, they focus on the fast migration between instances and the

high availability of services among a single class of mechanisms or even only among a single unified implementation.

We opted against a design that forces mechanism developers to adapt their implementation to a specific context representation but rather enable developers to integrate the existing mechanisms into our architectural concept. We offer a blueprint on how existing mechanisms can be integrated as well as guidelines on how newly developed mechanisms need to be designed to be context aware, and seamlessly contribute to a network wide state plane.

The introduction of transformation functions in STEAN further boosts the acceptance of such a context management system among network architects as well as mechanism designers. We not only use transformations to convert between different representations but contribute additional value to the state information gathered by the connected mechanisms. Our focus lies on the metadata that accompany the state information to form the network context.

The introduction of both a blueprint for designing and implementing transition aware mechanisms as well as a state plane architecture and implementation enables us to enhance the current state handling in networks to create a more dynamic and adaptive system architecture. It enables network architects as well as mechanism engineers to easily adapt existing mechanisms to our architecture, and quickly introduce new mechanisms that are built with transitions and context sharing in mind.

5 EVALUATION

Our system architecture is based on two major components, namely a context management system that enables us to transform and share context information between mechanisms in Section 4.1, and a guideline to design and implement transition aware mechanisms in Section 4.2.

In this chapter, we evaluate the individual components as well as the complete architecture by applying the principles to two distinct use cases selected from the scenarios presented in Section 3.1: "Reconfiguring Network Functions" and "Switching Routing Protocols".

While we already discussed the general system behavior of STEAN in Section 4.1.8, the analysis now focuses on the interaction of the individual components with the connected mechanisms as well as on the benefits we can gain from employing the architectural principles introduced in Chapter 4. We not only show how each individual component influences the system behavior but also discuss the implications of implementing the complete architecture, and how we are able to improve the performance, reliability and resource consumption of selected mechanisms.

First, we evaluate the behavior of STEAN as a context management system for migrating NFs. The dynamic scaling of NFs allows operators to provide the forwarding capacity required by the current network utilization without the need to over-provision resources. It allows to reduce the operational cost without loosing the ability to adhere to tight SLAs.

We compare our approach of a external context management and the continuous sharing of information to a framework that handles state locally and only transfers information when a migration is initiated. Our analysis is based on the implementation of STEAN presented in Section 4.1.6 on the one hand, and OpenNF—discussed in Section 2.5.5—on the other hand.

Our second evaluation is centered around the use case of switching routing protocols as introduced in Section 3.1.4. Existing WMNs use a static set of mechanisms and parameters that are chosen during deploy time and that do not necessarily offer the optimal performance under changing conditions. Our approach offers an exit route to this dilemma by selecting the appropriate mechanisms on demand and allowing a transition of the network during runtime.

We show that a transition of mechanisms in a highly utilized network can dramatically improve the forwarding performance both in terms of latency and jitter, and we demonstrate how employing our architecture can avoid the existing traffic gap when using a hard switch.

The use cases span a wide field of current network operations from the mechanisms employed in core networks and DCNs to state of the art client access networks using D2D communication. Our evaluation therefore covers a broad spectrum of current challenges in network operations and shows how all areas can benefit from a generalized context management system, and a mechanism architecture that is specifically built with transitions in mind.

5.1 MIGRATING NETWORK FUNCTIONS

In this section, we evaluate the behavior and performance of our proposed context management system when deployed in a NFV environment to handle the context of a group of NFs that provide both load balancing and failover services. Our goal is to compare context sharing using STEAN with current state migration systems for NFV such as OpenNF, and show that a generalized context management approach is feasible even for high performance mechanisms.

We use the migration of NFs—introduced as a use case in Section 3.1.1—to demonstrate the performance of STEAN and to discuss the advantages of a generalized context management system over a solution specifically tailored to a certain implementation of a NF.

We focus on the time required to successfully migrate the handling of network flows from one instance of the NF to a second instance without interrupting the packet forwarding. Additionally, we monitor the forwarding performance of both instances during the migration process and analyze the system behavior.

Network operations are usually bound to tight SLAs that define not only the general availability of the service but specify exact boundaries for critical network parameters such as forwarding performance, latency and jitter. These parameters need to be monitored and the network operator must ensure that the stipulated thresholds are not exceeded.

The compliant operation of the network can be achieved by either planning for the worst case, over-provisioning the required resources and thus underutilizing the infrastructure during normal operations, or by employing a dynamic scaling of NFs depending on the current load. The dynamic scaling frees resources on the underlying infrastructure such as VM hosts and allows to utilize the systems as needed.

Today, ISPs usually deploy proprietary and vendor specific solutions that allow for dynamic scaling during runtime, but are limited to products from the chosen vendor or even to certain implementations. This leads to a strong dependency on the chosen vendor and implementation, and disallows to combine the best currently available mechanisms. In other words, the operators run networks that leave a lot of room for improvement.

We thus introduce an open system that is able to not only handle the context of a single group of NFs or even a single vendor specific implementation but that can manage the context of various NFs and supplies multiple implementations with the required information. We are able to 1. overcome the vendor lock-in, 2. deploy the best system available for the current operation, and 3. exchange the NF with a feature equivalent replacement during runtime without any impact on the user or even service interruption.

In this evaluation, we are using a modified implementation of the PRADS asset monitor that supports OpenNF and also includes our extensions for STEAN support as described in Section 4.1.7. STEAN serves as the context management system that handles all flow related information while we utilize the existing functionality of OpenNF to initiate the migration. This separation allows us to directly compare the performance of existing state management solutions with our approach of external context handling.

We have chosen OpenNF as it is one of the most discussed state migration frameworks in the research community. The authors have proven that their design and implementation has no impact on the forwarding performance of the NF, that the time required for a migration is very low, and the implementation overhead is manageable. OpenNF therefore

serves as a reference for both the performance and the usability—in terms of ease of implementation.

5.1.1 *Experimental Design*

The design of our experiments is closely based on the requirements of a network operator running multiple instances of a NF that require migration support in order to provide fault tolerance and scalability. We focus on the problems that arise during the migration process and identify the relevant parameters influencing the migration performance as well as the metrics to show the performance of the proposed architecture.

The evaluation is done using a virtual environment—described in detail in Section 5.1.2—that allows us to control all parameters of both the network and the hosts running the NF instances. Therefore, it enables us to define multiple scenarios that typically appear in a DCN. We use the same general setup as proposed by the authors of OpenNF to facilitate the comparison of our results to those of the evaluation presented in [28] and enable other researchers to reproduce our results.

5.1.1.1 *Parameters*

The migration performance of a mechanism is influenced by three main parameters, namely 1. the network performance available for coordinating the migration and transferring the context information, 2. the size of the context information stored in either the migrate-from instance or STEAN, and 3. the number of currently connected flows to the migrate-from and the migrate-to NF. Therefore, we discuss these contributing factors in more detail and show how a change is influencing the measurement results.

NETWORK PERFORMANCE: The network performance is one of the critical parameters when migrating context or even using a context management system to externally store information. This includes both bandwidth and latency as the contributing factors influencing the migration performance of the complete system.

The bandwidth is relevant when transferring large amounts of data in a staked time frame—such as the migration of the complete context between NFs while the latency is an influencing factor when small parts are continuously requested, e.g., required information stored in a context management system.

Today's DCN architectures consist of two distinct networks, one comprising the data plane that handles the production traffic and one forming the control plane for managing the environment, including the orchestrating systems such as SDN controllers. The two networks often differ in performance by an order of magnitude and thus offer different capabilities for handling packets.

CONTEXT SIZE: The size of the stored context influences not only the total migration time as more information has to be transfered but potentially also the time required to migrate each flow. This is especially relevant for storing context in an external management system as these systems gather information not only from a single pair of NFs but from mechanisms running in a complete network segment and therefore need to handle a larger dataset.

The size of this dataset influences the response time of the context management system for retrieving specific entries as shown in Section 4.1.8 and thus contributes to the migration time of each individual flow. The size of the context again refers to both dimensions of the parameter: 1. the number of items stored within the context, and 2. the number of bytes required to store those items. While the storage size of each item is usually low, the number of items is large as multiple mechanisms contribte their information the the context management system.

NUMBER OF CONNECTED FLOWS: The number of connected flows defines the workload of each NF. It sets the baseline for the utilization of each instance and thus how much resources are available for handling the additional load introduced by the migration process.

Additionally, it also specifies the amount of context information changed during the migration, and which information requires special care to ensure that the current state is available on the migrate-to instance when the flow is transfered.

5.1.1.2 *Metrics*

We identified several metrics that allow us to assess the system behavior of both frameworks for direct context migration and transitions supported by a context management systems, and compare the results.

MIGRATION TIME: The *total migration time* $t_{migration} = \sum t_{flow}$, i.e., the time (in seconds) required to migrate the complete context information between NF instances as well as transferring the data path using SDN, is critical for the overall performance of the evaluated system. It forms the main metric for identifying the workload the system under test is capable of handling, and thus lets us anticipate if our solution is feasible for the use case.

Additionally, the *migration time per flow* $t_{flow} = t_8 - t_0$ (in seconds) provides us with insights on the behavior when the system is not running under maximum load. It represents the interval in which either all forwarding operations on a particular flow have to be suspended to fully migrate the context information, or all packets belonging to a flow need to be replayed to ensure order preserving operations.

It consists of the time required locally within each NF $t_{nf} = (t_1 - t_0) + (t_8 - t_7)$ as well as the time required to transfer information across the network $t_{net} = (t_2 - t_1) + (t_7 - t_6)$, and the processing time required by STEAN $t_{stean} = t_6 - t_2$. The migration time per flow allows us to infer the general conduct even in larger installations, and provides us with a baseline to identify the upper boundary in terms of packet and flow handling capacity.

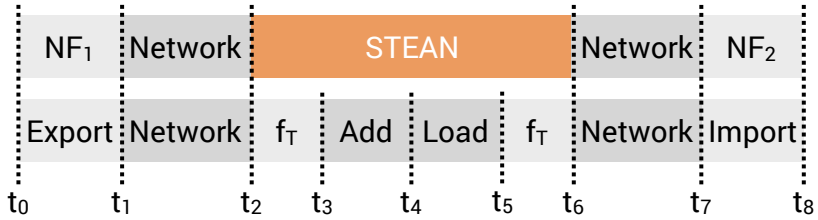

Figure 5.1: Migration time per flow split into subintervals.

STEAN TIMES: The time (in seconds) required by STEAN is critical for the overall system performance when employing a context management system. Therefore, we further divide the time t_{stean} into transformation time $t_{trans} = (t_3 - t_2) + (t_6 - t_5)$ and the time required by the storage backend $t_{store} = (t_4 - t_3) + (t_5 - t_4)$ to add and retreive the information. While the direct migration of context or the migration employing a controller to forward the information does not exhibit any delay due to storage and retrieval, using a context management system to handle the information always imposes a certain latency. This is due to access delays of the storage component and the execution of transformation functions. Therefore, we monitor these delays when employing STEAN to handle the context and the observations enable us to gain insights on the influence of our context management system regarding the total migration time.

SYSTEM LOAD: The system load (in percent) shows how much overhead the migration process imposes on the systems—either physical or virtual—hosting the NF instances. It offers insights on the efficiency of the transition as well as revealing potential bottlenecks for high load operations.

We are not only focusing on the load generated by the mechanisms but also examine the impact of running the orchestrating SDN controller as well as the context management system. The controller is critical for context migration using OpenNF as it resides on the critical path and all information must pass the single instance. The same applies for STEAN when running a context management system with the exception that the load is not suddenly applied when the migration is initiated but the system must be able to continuously handle the emerging load.

NETWORK UTILIZATION: The network utilization (in percent) is offering an insight into the transfer of information not only during the migration but also during normal operation. It is important to first establish a baseline to evaluate the additional overhead introduced by the migration process and to ensure that the network is not overloaded during the complete operation as such an overload would result in both bad NF performance as well as extended migration times.

We identified multiple parameters and metrics that are relevant for evaluating the performance of context migration between NFs. In the following experiments, we focus on the most relevant parameters that influence the time for a transition and evaluate the most significant metrics in terms of comparing the performance of a specialized migration framework and a generalized context management system.

5.1.2 *Experiment Execution*

We used Mininet [43] as a virtual environment to provide the network setup for conducting our experiments since it allows us to easily specify a network topology, condition network links, and start multiple OS containers. This framework enables us to perform the complete evaluation on a single host without the need for special SDN hardware such as switches or multiple physical servers that host the required instances of PRADS, the SDN controller and STEAN.

The setup allows us to use a low range server machine with a Quad-core Intel Xeon CPU and 16 GB of RAM to execute all experiments without any performance issues. However,

we bound each virtual host running the NF as well as the controller and STEAN instances to a dedicated CPU core to avoid context switches.

The architecture of our virtual network follows the principles of a SDN architecture, and separates the experiment platform into two distinct planes as depicted in Figure 5.2: 1. A data plane that handles all network flows, and 2. a discrete management plane that is responsible for handling the command and control instructions from the SDN controller.

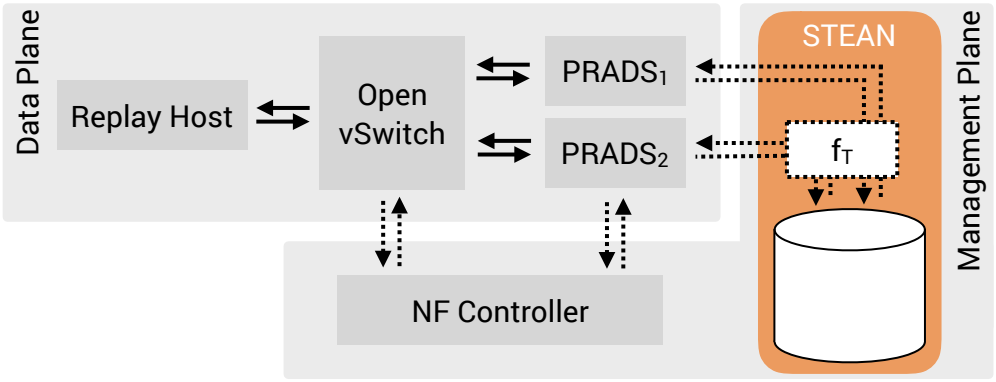

Figure 5.2: Experimental setup for the evaluation.

The data plane consists of two Mininet hosts that each run a dedicated PRADS instance ($PRADS_1$ and $PRADS_2$), and a separate host that is responsible for generating the workload by replaying a previously recorded network trace. All hosts are connected via a virtual 100 MBit/s link to an Open vSwitch that is responsible for segregating the network flows depending on the responsible NF instance.

On the management plane, we use Floodlight as the SDN controller—with patches from the OpenNF project applied—to enable the handling of state migration as well as the orchestration of the data path. The SDN controller is connected to the Open vSwitch as well as the PRADS instances via an unmanaged switch using 10 MBit/s links.

We have chosen the available bandwith based on the assumption that state of the art DCNs run at least 10 GBit/s links in the data and 1 GBit/s links in the management plane. Thus, the performance of the data plane is one order of magnitude higher than the performance of the management plane.

The workload on the data network is generated by replaying a network trace from a university-to-cloud scenario where multiple end systems within a local network connect to various cloud based services.

We used tcpdump [67] to record our own trace on the border gateway of our office network as the trace used by the authors of OpenNF is not publicly available. The trace was recorded over a period of approx. 20 h, and contains 70 k TCP flows, 2/3 of which are HTTP(S) flows generated by connecting to typical web services such as Dropbox, Google Calendar and Youtube. Only a minimal amount of these HTTP(S) flows were generated by users actually browsing on websites. The application mix of our trace with more than 60 % of HTTP(S) is also observed at large IXPs [4, 61] as well as on residential lines [48], and thus is exemplary for a typical workload of NFs.

On average, a flow has a duration of 35.0 s and 5.0 flows are active in parallel. For 13 % of the time, more than 100 flows are active in parallel. The mean number of flows is 33.6 which comes from a few long lived management connections that are active during the whole capture.

We determined the maximum packet rate our setup can handle experimentally and found that the processing of flows on the NF as well as the migration process were unreliable when more then 550 packets per second are emitted by the host replaying the trace. Therefore, we chose to use a sending rate of 500 packets per second to not operate our network under maximum or even overload but to leave some headroom for additional flows.

All migrations are executed with order preservation enabled as it is one of the critical requirements for operating NFs to not generate false context during the migration and thus potentially raise false alerts, e.g., on a firewall or IDS.

The Open vSwitch initially redirects all traffic to $PRADS_1$. Once the NF instance has created state for 250 and 400 flows, respectively, we initiate the migration of the flow state to $PRADS_2$. We have chosen these thresholds due to the limited packet buffering capacity of Floodlight when operated in order preserving mode.

While it seems more efficient to provide the context of all flows to $PRADS_2$ at once, STEAN executes filter transformations to select the context of the flow that is currently migrated. This limits the additional load on the data plane to the absolut necessary information at any point in time.

5.1.3 *Results*

We now present two distinct aspects of our evaluation, and undermine our findings with selected results from our extensive series of experiments.

First, we put the focus on the overall system behavior during the migration process, including all components from the two PRADS instances serving as an example for a virtualized NF, over the extended SDN controller that orchestrates the migration process, to STEAN as the backend for context management. This allows us to provide a direct quantitative analysis of the migration performance employing our proposed solution in comparison to state of the art migration frameworks.

Second, we analyze the behavior of STEAN during the migration process, and how the prototype is handling the imposed load. This allows us to provide a qualitative analysis of our architectural principals in general and the introduction of a dedicated state plane in a DCN environment in particular.

For a general system analysis, we compare the OpenNF implementation to our extended framework using STEAN as a context management system. The box plots shown in Figure 5.3 present the results for two different workloads that 1. resemble a similar workload to the one used by the authors of OpenNF, and 2. represent the maximum load our current setup can handle. In Figure 5.3, we show the migration time for one TCP flow as the most relevant metric for the forwarding delay caused by the migration process.

We observe that employing STEAN for context migration reduces the median migration time per flow from 103 ms to 41 ms for the 250 flow case, and from 280 ms to 160 ms for the 400 flow case. This equates to an improvement of the median migration time between 42 % and 60 %, where a larger improvement is achieved when fewer context is stored.

The results also show that OpenNF completes its operation in a narrow time frame while employing STEAN results in a higher variability. For the 250 flow case, we note that the third quartile for both systems is at 145 ms and the first quartile is about 50 % lower for STEAN. The distance between the first quartile and the median indicates that STEAN is able to respond to most requests in a narrow time frame, and the larger variance results

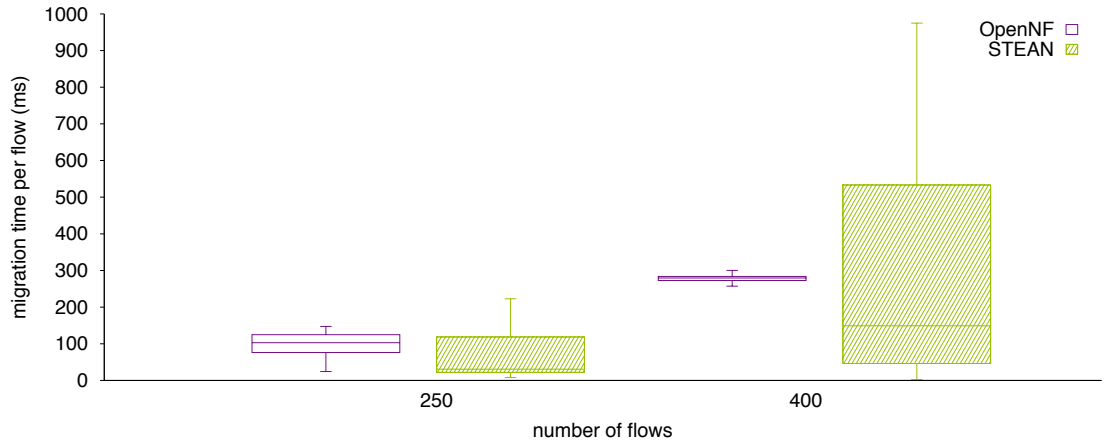

Figure 5.3: Total migration time per flow context between two PRADS instances.

from some operations that take significant more time but are still in the scope set by OpenNF.

This is even more visible at the results for the 400 flow case where both the first quartile as well as the median for STEAN are far below the results for OpenNF. However, OpenNF is able to response in an even narrower range while the variability—when using STEAN—is further increased.

The higher variability is mainly due to database locking of the current STEAN storage backend on inserts and updates. The current implementation does not allow for concurrent operations when executing write operations on the data set and thus delays all concurrent requests until the insert or update is completed.

In the median case, we observe that running STEAN increases the migration performance, and the usage of a dedicated context management system only comes with a negligible overhead when migrating TCP flow state between PRADS instances. This additional overhead for some operations is represented in the increased variability for the migration time of a single flow.

Increasing the number of flows to be migrated above 400 results in a large increase in time between storing the context on PRADS_1 and retrieving the context on PRADS_2. Therefore, we specifically analyze the performance of STEAN during these operations to demonstrate where potential bottlenecks are located.

The results presented as box plot in Figure 5.4 show the response time of STEAN for the 250 and 400 flow case examined before as well as the results gathered from experiments migrating 1000 and 1500 flows respectively.

The evaluation shows that the times for operations involving STEAN remain almost constant even though we observed a dramatic increase in total migration time up to the point where the operation of transferring 1000 and 1500 flows were never completed.

Additionally, we analyzed the system behavior using various bandwidth settings from 10 MBit/s to 50 MBit/s for the control network in order to eliminate the possibility of congestion during the migration.

However, the different bandwidth settings did not have any effect on the migration time for both setups—OpenNF and STEAN—which shows that the network is able to provide sufficient resources even when using a lower bandwidth. Thus, the overall

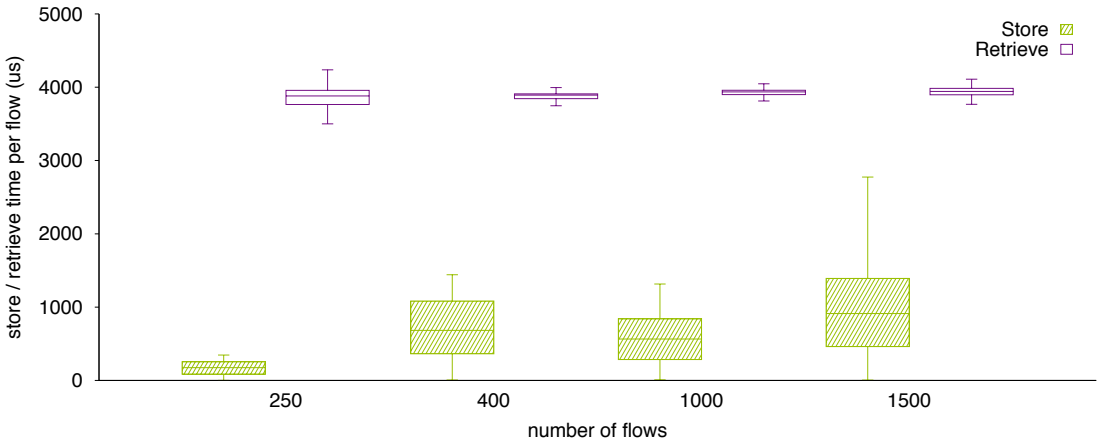

Figure 5.4: Store and retrieve time per flow for migrating flows using STEAN.

performance decrease when more than 400 flows are migrated is not due to a bottleneck in STEAN but originates from an overload of the controller supplied by OpenNF.

5.1.4 *Summary*

In this section, we evaluated the system behavior of STEAN when employed as a context management system for NFs in a data center environment, and we showed how the direct sharing of context information can improve the migration process. The experiments were conducted using a virtual network environment using Mininet. We replayed a previously recorded university-to-cloud trace to two instances of PRADS configured in a migration scenario.

The focus of our evaluation was on the time required to successfully transfer the handling of network flows from one instance of our NF to another without interrupting the traffic or pausing the processing of packets. The migration also included the transfer of context information required by the NF such that consecutive packets belonging to the same flow are identified and classified accordingly.

Our results show that sharing context using STEAN is generally faster than migrating state employing OpenNF. This increase in performance mainly results from the improved sharing of context using STEAN where information is directly accessed by the requesting NF. While OpenNF only shares information between instances when the migration is initiated and otherwise operates the mechanisms completely independent of each other, our approach allows for a continious exchange of context between running mechanisms.

The global view on all context information provided by STEAN along with the active communication between the context management system and the connected NFs enables us to migrate the flows between instances without the need to first transfer the required state information. We are able to directly serve the context associated with the migrated flow to the now responsible instance, and thus avoid the otherwise required state transfer.

The direct exchange of information also allows us to remove 1. the SDN controller from the critical path, and 2. the need to buffer data packets to provide order preserving delivery. In our architecture, the controller is only responsible for orchestrating the migration process and does not handle any data traffic which significantly reduces the load on the controller and improves its responsiveness to migration requests.

Additionally, our network design allows for further enhancements in the migration process as the context information could now be transfered using the faster data plane. While the data plane carries a much higher load than the control plane, the better network performance both in terms of bandwidth and latency could help us to reduce the migration time. Especially the reduced latency is a key factor as the response time of STEAN highly depends on the delay of the network connection to the mechanisms. Thus, we are able to deliver the requested context information faster and improve the overall responsiveness of the system.

However, we also observed that the variability of the migration time per flow increases. This is mainly due to our current implementation of the storage component which blocks all other operations during a write access to avoid the delivery of inconsistent information. When one instance of PRADS writes context information to STEAN, all requests of the other instances are delayed until the write operation is finished, therefore increasing the response time of these requests and subsequently the migration time of the associated flows.

One possible solution to this limitation of our current implementation is to partition the XML document managed by the storage component into multiple subdocuments that can be locked independently and thus reduce the amount of data blocked by each write operation to a minimum. This partitioning is already done in Relational Database Management Systems (RDBMSs) where multiple tables or even single rows can be independently locked for write operations [8].

In this evaluation, we have shown that the direct sharing of information using STEAN as a context management system can compete with state of the art systems for migrating NF state, and even improves the overall migration performance of the system. Our architecture not only centralizes the context information associated to each flow for direct access by the processing NF but also enables us to improve the network design in terms of utilizing the best available network connection to handle context requests, and remove the SDN controller from the critical path during migration.

5.2 SWITCHING ROUTING PROTOCOLS

In this section, we evaluate the feasibility of our system architecture for the use case of switching WMN routing protocols presented in Section 3.1.4. We show that 1. our approach is able to handle the context information of WMN routing protocols even on low powered nodes, and 2. we are able to execute mechanism transitions using a hard switch while precluding a gap in end to end connectivity. Our focus is on the applicability of transformation functions to provide seamless transitions between mechanisms using STEAN.

While it is possible to avoid the traffic gap—inherent to a hard switch—by operating multiple mechanisms in parallel to bootstrap the context information required by the new mechanism, this soft switch imposes additional overhead on the network as discussed in Section 3.2. Using a context management system such as STEAN, it is possible to reduce the time of parallel operation, required to avoid the traffic gap, to a minimum or—depending on the specific mechanism properties—even use a hard switch and thus completely eliminate the need for parallel operation.

It is generally desirable for network operators to minimize the time required for a transition and thus the time of parallel execution as multiple subsequent transitions can

be executed on a tighter schedule. This leads to a more flexible and dynamic network that can faster react to changes, and cedes more network resources to the actual data traffic.

However, the reduction or even complete avoidance of parallel operations is especially important in environments where the transition is only invoked when a problem in the network arrises as it is too costly to precautionary switch mechanisms. These networks often only have scarce resources and are highly utilized which both prohibits the additional overhead of running multiple mechanisms in parallel and forces the operators to reduce the time required for the transition.

Additionally, the necessity of invoking a transition in such a network implies that there is already either a high utilization or the network is currently not capable of providing the intended throughput. Thus, putting additional load on the system by operating mechanisms in parallel often exacerbates the problem instead of providing a viable solution.

In this evaluation, we employ STEAN to manage the context of the WMN routing protocols and transform the routing information between mechanisms during the transition. Our main goal is to show how transformation functions can be employed to share context information between different protocols and how the shared information can be used to allow for a seamless switch between these mechanisms during runtime.

Furthermore, we show that using STEAN as a context management system only imposes a minimal overhead on the forwarding delay even on low end WMN nodes, and that it is feasible to store all context in STEAN during normal operation. The directly shared context information allows us to use a hard switch to transition between routing protocols while providing seamless network operation during the transition phase with only slightly increased end to end delay.

While we chose WMN routing protocols as an example, our aim is not to present a performance evaluation of different WMN routing protocols. Instead, we focus on the effects during a transition of communication mechanisms.

5.2.1 *Experimental Design*

During the experiments only selected nodes were active and participating in the mesh network. We used this setup to force the protocols to choose specific routes and to create a bottleneck in the network to achieve a maximum gap in network traffic when executing a hard switch without context sharing. This leads to route lengths of two and three hops respectively as shown in Figure 5.5.

The experiments are conducted with various packet rates between 10 and 250 packets per second (pps), and packet sizes of 300 byte to 1000 byte. We expect the total length of the gap in end to end connectivity to be dependent on both parameters—along with various protocol dependent parameters discussed in Appendix A—while the performance of STEAN does not depend on the absolute throughput of the network. It is solely dependent on the packet rate as the number of requests to STEAN does not increase when using larger packets.

Our main metric for evaluating the network performance is the end to end delay (in seconds) of a single connection between a pair of nodes. We have shown in Section 4.1.8 that STEAN introduces a latency penalty as additional operations such as network calls, transformations and storage lookups have to be performed. Thus, using STEAN increases

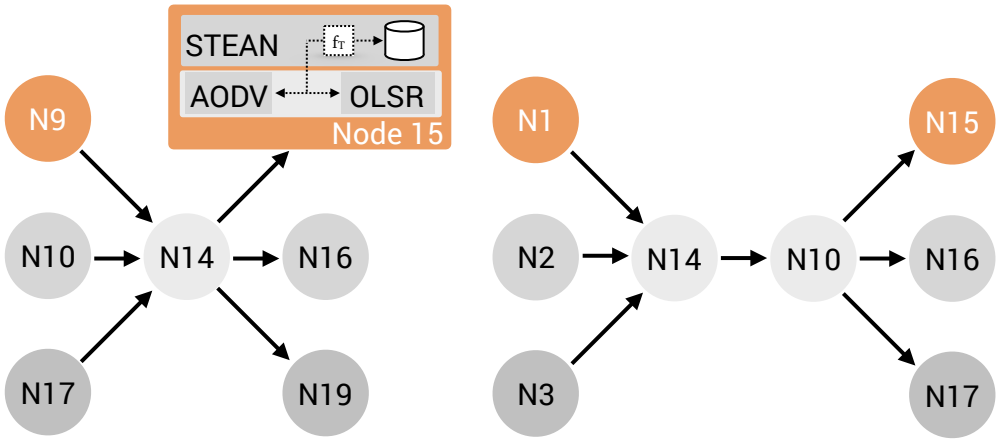

Figure 5.5: Communication setup with two and three hop routes. The enumeration represents the nodes as shown in Figure 5.7. All all nodes are STEAN-enabled, and run our modular implementation of AODV and OLSR (exemplary shown for node 15).

the forwarding delay when used as a context management system for routing information while the other network characteristics such as throughput are not affected.

Additionally, we use the remaining node pairs to generate background traffic that increases the load on the intermediate hosts where the packets of all flows have to be processed. Therefore, the background traffic has a direct influence on the delay of packets in the observed flow, the system load of the intermediate node as well as the size of the traffic gap when a mechanism switch is executed.

However, we ensure that neither the nodes nor the routing protocols are overloaded and only analyze the behavior of the components in a stable state—that is when all routes are established and a continuous flow of packets is guaranteed. Our goal is not to examine the mechanism behavior during route establishment and network interference but the system characteristics of STEAN when serving as a backend for sharing routing information.

One of the core factors influencing the end to end delay is the Round Trip Time (RTT) between STEAN and the connected mechanisms as each packet generates at least one lookup to the context store for determining the next hop on the route—unless client side caches are in place—to ensure that subsequent requests can be answered locally. These requests add additional delay multiple times during an end to end transmission as the round trips have to be done on the source node and each forwarding node.

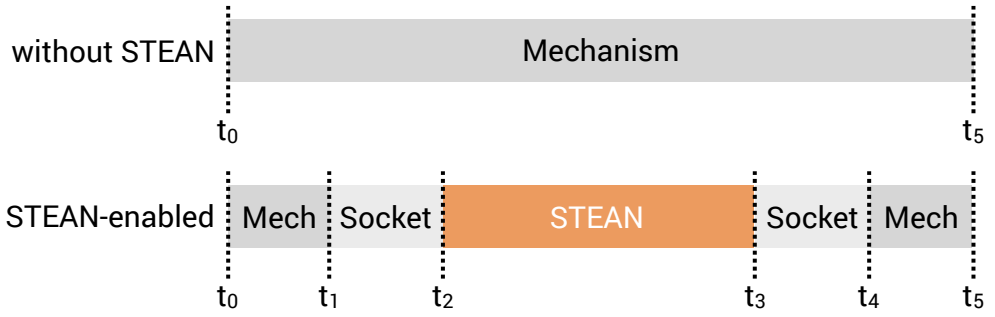

Figure 5.6: Local forwarding time split up in subintervals.

We can further divide this *local forwarding time* for each node into subintervals as shown in Figure 5.6. We define the *mechanism time* as the time a request is solely processed by the mechanism implementations. This excludes the intervals for OS operations such as receiving a packet on the inbound network interface and sending the packet on the outbound interface after local processing as well as the time a mechanism is waiting for a response from the context management system. The mechanism time is determined by $t_{mech} = (t_1 - t_0) + (t_5 - t_4)$. The *socket time* describes the time spent on (local) network communication between the mechanisms and STEAN to send the request and receive the corresponding response and is calculated as $t_{sock} = (t_2 - t_1) + (t_4 - t_3)$. We refer to *STEAN time* as the interval between receiving the request in STEAN and sending out the response. This interval includes all operations within STEAN, e.g., dispatching the request, database lookup, applying transformation functions and assembling the XML response and is determined by $t_{STEAN} = t_3 - t_2$.

For mechanisms not connected to STEAN, we assume that $t_{sock} = t_{STEAN} = 0$ The total local forwarding time is then calculated as $t_{fwd} = t_{mech} + t_{sock} + t_{STEAN}$ for each instance.

We also monitor and evaluate the resource usage of the involved software components—namely the OS, Click and STEAN—during the experiment execution. Therefore, we record the CPU utilization (in percent) as well as the RAM usage (in percent) of each component over time to 1. ensure that the nodes are not overloaded, 2. gain insights on the resource usage of each component, and 3. analyze the impact of using a central context management system on the host system.

We compile and execute Click as a single threaded program in user space to avoid the tedious setup of kernel mode which might provide insignificant performance gains, but disallows the use of C++ Standard Library (STL) functions.

However, STEAN is compiled multi threaded in order to reduce the response time for multiple parallel context requests as well as to enable parallel execution of the maintenance tasks.

We specifically omitted experiments running the secured version of SAODV and SOLSR as previous experiments—presented in [74]—showed that the nodes available in our testbed were not capable of performing the required cryptographic operations in real time.

5.2.2 *Experiment Execution*

We use our WMN testbed currently consisting of 17 statically mounted wireless nodes across two floors in an office and lab building to conduct the experiments. The location of the nodes within the building is depicted in Figure 5.7.

Each node is composed of a x86 compatible 500 MHz AMD Geode LX800 CPU with 256 MB of RAM and Atheros wireless cards. The CPU only has a single core which precludes the parallel operation of the Click and STEAN process as well as the parallel execution of multiple STEAN threads. The nodes also have a wired ethernet connection for booting from a remote server and for receiving control messages during the experiments. We run an off-the-shelf Linux with Kernel 3.13.5, and use tcpdump on each node to capture the wireless traffic.

The nodes run our Click based routing protocol implementation presented in Section 4.2.3 along with the extensions presented in Section 4.1.7 to connect the mechanisms

Figure 5.7: Node deployment in our testbed experiment.

to STEAN. This setup allows us to use the same mechanism implementation in all experiments, and we only need to replace the context storage element when necessary. To keep the overhead and thus the influence of the not running mechanism to a minimum, all timers are turned off by default and only get activated when the mechanism is in use.

Additionally, each node runs an independent instance of STEAN that has the base context and the transformation functions described in Section 4.1.7 configured. The local STEAN instance ensures that each node runs independent of the network connectivity and resembles the operational mode as well as the behavior in a WMN as close as possible, despite the fact that a wired out-of-band connection is available. We also enabled client-side as well as STEAN-side caches for optimal forwarding performance.

We utilize the command and control infrastructure presented in Section 4.2.3 to manage the node configuration, control the experiment, and trigger the transition.

The control server connects to each node using the wired out-of-band link to not interfere with the measurement traffic. This setup also ensures a reliable control channel even when the wireless medium is under heavy use and thus prevents a service interruption due to different execution states. It leaves the reestablishment of routing information as the only source for a potential traffic gap.

The Distributed Internet Traffic Generator (D-ITG) [12] is used to generate the workload and measurement traffic. The software offers an easy way to specify the exact payload length as well as the sending rate. Thus, the used bandwidth can be easily determined. Additionally, D-ITG allows to measure the end to end delay by simply logging the sending and receiving time of a packet and then calculating the delta. As this measurement highly depends on accurate clocks on all nodes, we use the NTP to sync the time on all nodes.

Even though the nodes keep a permanent connection to the control server, the measurement data was collected locally on each node, and only mirrored to the central storage after each experiment run was finished. This ensured that the network stack of the node was not utilized by auxiliary tasks such as NFS traffic.

5.2.3 *Results*

First, we discuss the general behavior of the routing protocols without a mechanism switch.

The time series graph in Figure 5.8 shows the end to end delay on a two-hop route in our testbed running OLSR for both the original implementation as well as our modifications that enable context sharing. In these experiments, we transmitted 250 pps with a packet size of 1000 byte in a single flow which results in a total sending rate of 2 MBit/s.

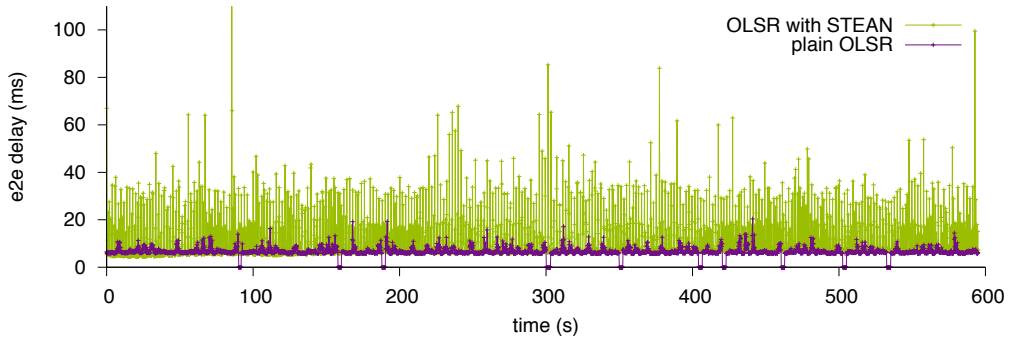

Figure 5.8: End to end delay running OLSR on a two-hop route.

The average delay for the unmodified mechanism over our 600 s experiment runtime was 6.48 ms with a maximum delay of 20.36 ms, and a jitter of 1.88 ms. Additionally, we observed a packet loss of 4.2 % which most likely results from unstable wireless links caused by interference with other transmissions on the same channel.

Our analysis of the CPU load and RAM distribution between the various processes running on the node showed that most of the RAM was used by the tcpdump process for capturing the wireless traces and the Click process did only consume 1.4 % of the total memory and generated a processor load of 7.8 %.

The results—summarized in Table 5.1—indicate that the wireless links are not in an oversaturated state as otherwise buffers and therefore RAM usage would increase. Thus, the nodes' hardware is capable of handling the imposed load. Thus, the packet size and sending rate chosen for our evaluation is a good baseline for evaluating the network behavior when using STEAN as a context management system on each node.

PARAMETER	CLICK		STEAN
	WITHOUT STEAN	WITH STEAN	
CPU	7.8 %	10.1 %	13.1 %
RAM	1.4 %	1.5 %	1.1 %

Table 5.1: Performance metrics on a single node while running OLSR.

For the experiments that use OLSR with STEAN support to manage the mechanism context, we observed an increase of the average end to end delay to 11.93 ms and an increase of the jitter to 10.15 ms but recorded no packet loss. Measurements of the hardware utilization showed an increased CPU usage for Click to 10.1 %—due to the additional connection required to contact the context management system—and a memory usage of 1.5 % while STEAN used 13.1 % of the computational resources and 1.1 % of the RAM available.

While the increase in both delay and jitter seems large at first, it is still acceptable for almost all applications running across a WMN and even allows for Voice over IP (VoIP) calls [36]. We also have to take into account that the network is running under optimal conditions where delay and jitter of the wireless link is minimal, and thus the effects introduced by STEAN are more significant. For networks with increased forwarding delay and jitter, the relative performance impact of utilizing a context management system is reduced. This holds especially true as the performance of our prototype is not bound by CPU or RAM limitations but by the throughput of the local network socket used to connect the clients.

The time series graph in Figure 5.9 shows the RTT of a STEAN request on a single node during another experiment with a similar workload. Here we can see that about 45 % of the total forwarding time (t_{fwd}) on a single node is spent on socket communication (t_{sock}) between the mechanism implementation in Click and STEAN while the—potential complex—operations within the context management system (t_{STEAN}) are responsible for another 45 % of the RTT. The computation of the actual forwarding decision within the mechanism (t_{mech}) is only responsible for 10 % of the total forwarding time.

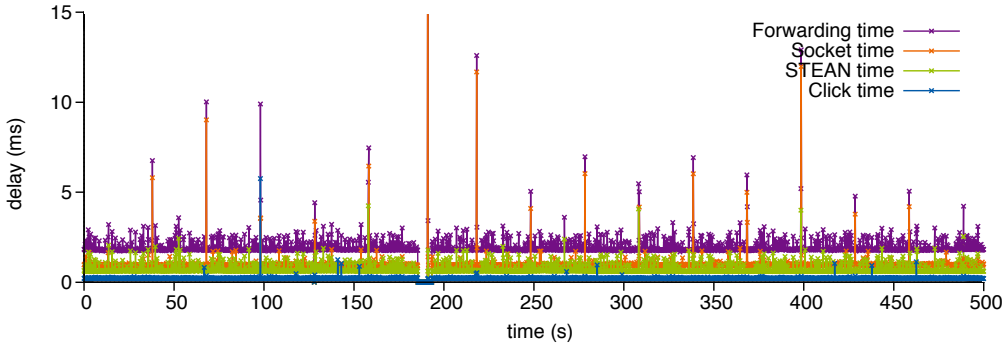

Figure 5.9: Local forwarding delay on an individual node over time for STEAN–enabled OLSR.

Additionally, we observed that for 2/3 of all packets the forwarding time is equal for both the standard and the STEAN-enabled implementation as shown in Figure 5.10 which presents the Cumulative Distribution Function (CDF) of the end to end delay running OLSR. The higher delay for other packets is due to blocking updates of the routing table that include packet counters, and thus is altered regularly. This shows that the performance decrease is not a general limitation of utilizing a context management system to share information across routing protocols, but can be addressed in optimized implementations of both the mechanism and STEAN. We could either store the counters locally within the mechanism implementation and relinquish from storing all context information—especially the information with a short life time and marginal use to other mechanisms—within STEAN, or we could enhance our implementation of the context

management to support concurrent add and modify operations by substituting the storage component.

The CDF also shows that 95 % of all packets arrive within 33 ms and the high jitter comes from a few outliers which are due to the locking storage backend. Therefore, the optimization discussed above would also reduce the jitter and further improve the forwarding performance of the modified routing protocols.

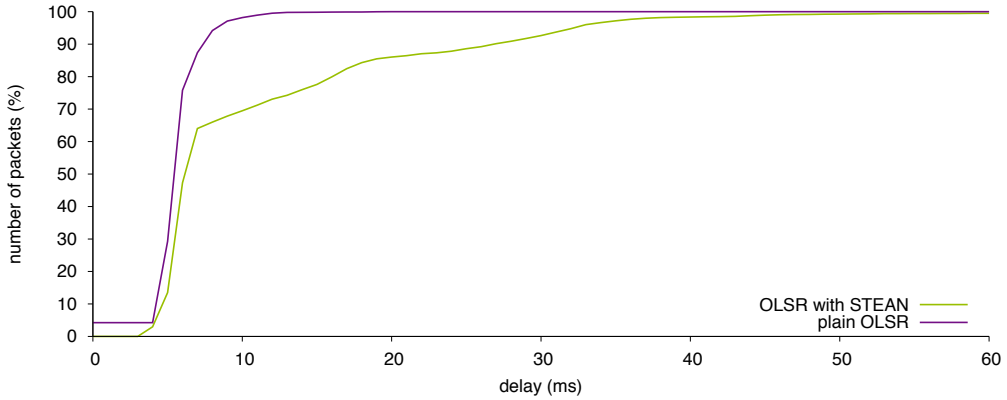

Figure 5.10: Cumulative distribution function of the end to end delay running OLSR.

However, disabling the STEAN-side cache dramatically degrades the network performance of a node, and the local forwarding time per node increases from 2 ms to 7 ms as presented in the time series graph in Figure 5.11 (using the same workload as in Figure 5.9). This shows that the caching infrastructure provided by STEAN is very effective, and significantly reduces the response time for subsequent requests even though we opted against a shared cache between all clients and use a separate cache instance per mechanism instead.

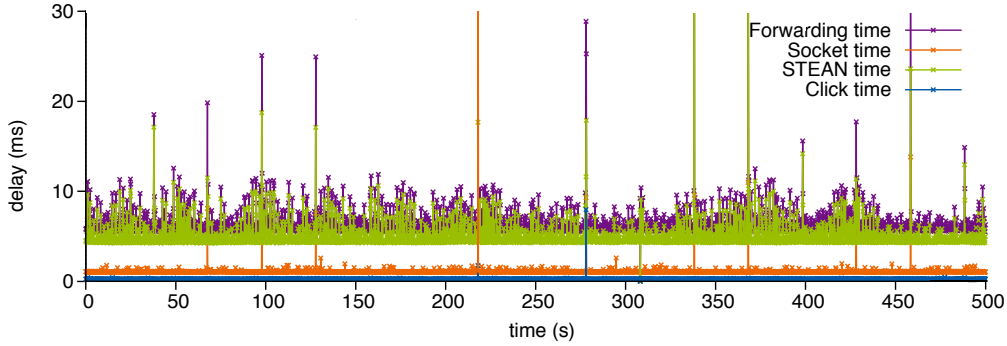

Figure 5.11: Local forwarding delay on an individual node over time for STEAN–enabled OLSR without caching.

Second, we show the results obtained from our evaluation of mechanism transitions during runtime using a hard switch. We present selected sample runs from our data collection that show the effect of different protocol parameters and workloads before presenting our findings on how a shared context is influencing the forwarding performance during the transition.

During each experiment, the transition was initiated after the running mechanism was in steady state, i.e., all routes required for end to end connections were established and the nodes included in the experiment were continuously forwarding packets.

The time series graph in Figure 5.12 shows the gap in traffic on a three hop connection with a packet size of 300 byte and a sending rate of 50 pps. The background traffic consists of two flows with the same parameters as the measurement connection. We ran OLSR as the initial mechanism and switched to AODV.

The total sending rate within the network is 351 kBit/s which is significantly lower than the 2 MBit/s that mark the stable thoughput our network can handle. Nevertheless, we observed a gap in traffic of 171.7 s during the switch.

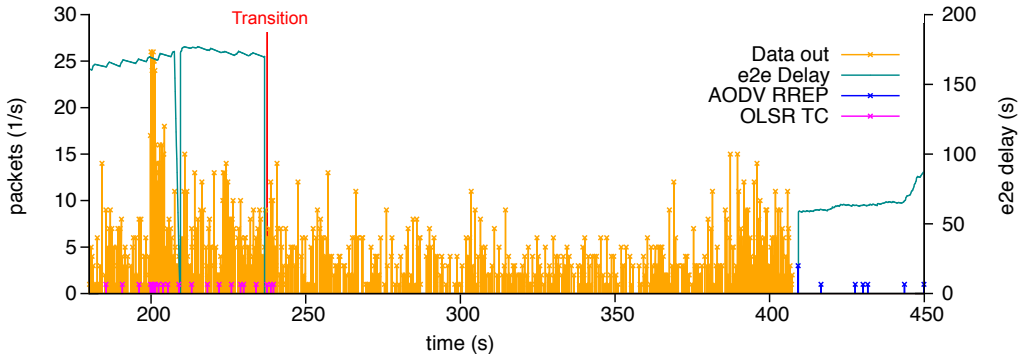

Figure 5.12: Traffic gap during the execution of a mechanism switch on a highly utilized network. A end to end delay of 0 indicates that no packets were forwarded during this time interval.

Additional experiments with a total sending rate of more than 400 kBit/s—only 1/5 of the possible stable throughput—showed that the network is not able to recover from the connection loss and that all management packets sent by the routing protocols during the transition are lost.

In the time series graph in Figure 5.13, we show a switch on an almost idle network running three flows with a packet size of 300 byte and a sending rate of 10 pps each or a total sending rate of 70 kBit/s. The experiment is launched running AODV and then the routing protocol is switched to OLSR. The protocol parameters in this setup are optimized for a higher mobility of nodes, and thus the protocols react quicker to changes in the network such as a switch in mechanisms, but also generate more traffic overhead due to the increased number of coordination messages. Even though there is only a low load on the network a gap of about three seconds is still visible.

We now use STEAN to share context information between the two mechanisms and thus provide the routing information already gathered by the preceding mechanism to the starting protocol.

The experiments are executed using a single sender that emits 250 pps with a packet size of 1000 byte or a total sending rate of 2 MBit/s. We have chosen this workload as our experiments showed that both the mechanism implementation as well as STEAN are capable of handling that workload, and a gap in network traffic is already visible with even a significant lower sending rate.

The forwarding behavior of our network during a mechanism transition from OLSR to AODV during runtime is presented in the time series graph in Figure 5.14.

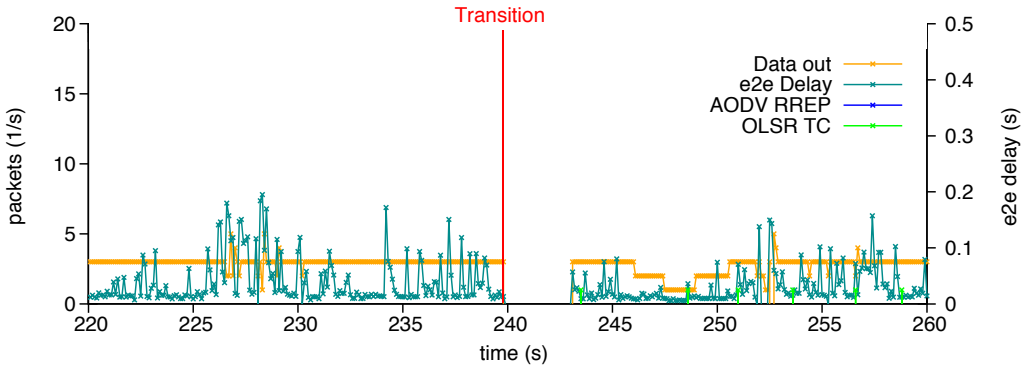

Figure 5.13: Traffic gap during the execution of a mechanism switch on an almost idle network. A end to end delay of 0 indicates that no packets were forwarded during this time interval.

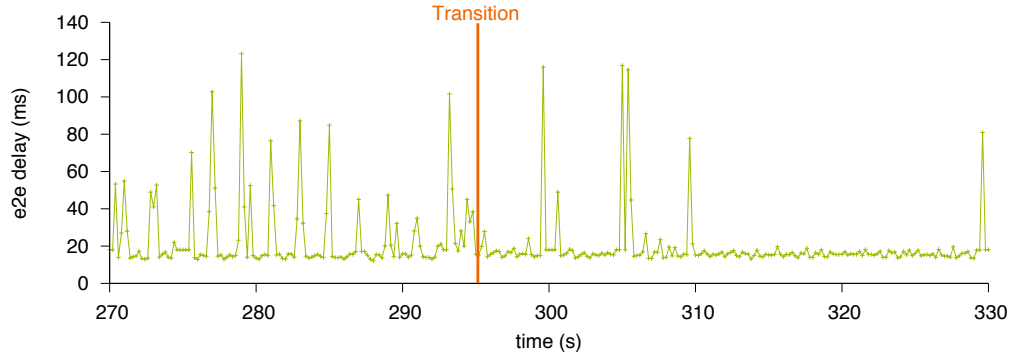

Figure 5.14: End to end delay when migrating from OLSR to AODV during normal network operation. The transition is executed after 295 s (red line).

The results show that the transition is executed without interruption in packet forwarding and the only visible effect is that the jitter is reduced from 16.46 ms to 4.26 ms and the average end to end delay drops from 22.95 ms when running OLSR to 18.15 ms when AODV is employed as a routing protocol.

We also observed that only a minimal number of AODV control messages such as HELLO, Route Requests and Route Replies are sent directly after the transition but the new mechanism almost solely operates on the existing context information—list of neighboring nodes, next hop to destination, existing network links—provided by the preceding mechanism without any service degradation.

5.2.4 *Summary*

In this section, we demonstrated the feasibility of our approach for sharing context information between WMN routing protocols by 1. showing that the performance impact of using a context management system such as STEAN is perceptible but not prohibitive even on low powered nodes, and 2. we are able to remove the effects of a hard mechanism switch during runtime by sharing the context information between multiple routing protocols.

Our focus was not only on the quantifiable metrics such as end to end delay and local forwarding performance, but also on the feasibility of sharing context information

between different mechanism designs—namely proactive vs. reactive routing protocols—in order to eliminate the traffic gap when switching mechanisms.

We used our modular implementation of AODV and OLSR that follow the guidelines presented in Section 4.2, and employed STEAN with integral support for transformation functions—introduced in Section 4.1—to accommodate the requirements of the used mechanisms. This provides us with a system to dynamically choose the routing protocol during runtime without the need to extensively modify the mechanism implementation or even add specialized data structures that support the requirements of all employed mechanisms. Instead, we are able to provide each mechanism with its preferred context representation and, at the same time, gather monitoring information from the internal structures of the mechanisms to support our switching decision.

Our system imposes an acceptable overhead on the network performance which is mainly visible in an increased end to end delay. We analyzed the behavior of all involved components and came to the conclusion that this increase is mainly due to the prototypical implementation of our architecture and the major factors responsible for to the overhead—namely the delay in socket communication and the response time of STEAN—can be improved by future implementations.

The evaluation shows that a traffic gap exists when switching between different routing protocols in a WMN and that this gap is not negligible but has a significant impact on end to end connectivity. It not only prevails if the load on the network is high but also when using low to moderate data rates. Thus, a mechanism transition effects all types of network operations from low bandwith text based applications over voice communication to video transmissions.

We conclude from the experiments shown above that a common context management in conjunction with transformation functions—as implemented by STEAN—along with the support of transition aware mechanisms—designed and implemented using the proposed blueprint—is able to provide a seamless mechanism switch in WMNs with acceptable overhead and enables mechanism transitions during runtime without loosing end to end connectivity.

Furthermore, our results show that the dynamic adaptation of mechanisms is a core requirement not only for optimizing the performance of communication networks but—during certain operational conditions—ensures the availability of end to end connectivity. The complete support for transitions in the employed mechanism along with integral support for context sharing are key enablers to achieve this goal.

Executing a switch of mechanisms that support context sharing is transparent to end systems as well as upper layer mechanisms and thus enables network operators to dynamically choose mechanisms during runtime without any negative effect noticeable by the users.

5.3 DISCUSSION

In this chapter, we presented the evaluation of our system architecture using two distinct use cases. We focused on the interaction of the individual components with the connected mechanisms as well as on the benefits we can gain from employing the architectural principals. Our goal was to demonstrate the feasibility of our approach and identify the deficiencies when using a centralized context management system such as STEAN.

The chosen use cases cover the complete spectrum of network operations and access technologies from employing NFV in a DCN or carrier environment to providing access services for end systems using WMNs. Thus, the use cases are suitable to evaluate a generalized context management system that is designed to serve the requirements of all mechanisms in a network as well as demonstrating the feasibility of our architectural concepts across all networking domains.

First, we analyzed the performance of STEAN during the migration of NFs by employing the extension described in Section 4.1.7 and compared its performance to the OpenNF framework. Our implementation uses STEAN as a backend for context management and allows the direct sharing of information between the NFs while OpenNF uses the data structures provided by the mechanism implementation and extracts the context when the migration is initiated.

The results show that the direct sharing of context is generally faster than the solution provided by OpenNF even though we apply transformation functions to the context information instead of directly attaching the serialized data structures to the migration command, and we add additional latency due to the required network communication.

Additionally, the direct sharing of information allows us to remove the SDN controller from the critical path and enables us to exchange the context information using the higher performance data plane instead of the management plane.

Second, we evaluated STEAN along with the implementation of our mechansism framework introduced in Section 4.2. The experiments are performed in a WMN testbed by executing transitions between routing protocols, and the evaluation is based on the use case presented in Section 3.1.4.

We analyzed the end to end delay and local forwarding performance of our implementation and showed the feasibility of employing transformation functions to share context information between different mechanisms.

The results demonstrate that employing transition aware mechanisms in conjunction with a context management system can completely eliminate the traffic gap that exists when switching mechanisms. Our prototypes inflicted an acceptable penalty on the forwarding delay that still allows the execution of real time applications such as VoIP calls. The increased delay mainly results from the additional communication overhead between the routing protocols and STEAN to exchange the context information, and can be reduced by optimizing the communication path.

We also showed that our approach is feasible even for low powered devices such as the nodes in our testbed. We show that STEAN is able to run on smartphones and embedded devices without any significant performance impact, and demonstrate the efficiency of our prototype despite the fact that some drawbacks remain.

We conclude from the results gathered from both experiments that 1. direct sharing can improve the migration performance of NF context, 2. the sharing of information between WMN routing protocols completely removes the otherwise existing traffic gap when switching mechanisms during runtime, 3. employing transformation functions to serve the requirements of different mechanisms impedes an acceptable overhead, and 4. the strict adherence to the proposed architectural principles enables network operators to easily deploy new mechanisms in an existing network.

While we focused on the system behavior during the migration of context and the transition of mechanisms, our experiments also show that our prototypical implementation

generally reduces the forwarding performance of a mechanism but that this penalty is negligible.

The analysis shows that our approach is feasible for a wide variety of use cases and allows us to not only handle the context of multiple mechanisms but also to share that context across these mechanisms. Our prototype is capable of handling the workload applied during the experiments, and all limitations during the evaluation were caused by either the host system or the available network links.

Thus, we argue that the direct exchange of context information between mechanisms using a context management system such as STEAN in conjunction with architectural guidelines to develop and implement mechanisms leads to an improvement in migration performance, network reliability, and operational overhead.

6 CONCLUSION AND OUTLOOK

In this thesis, we have introduced distinct use cases that benefit from extended context sharing across mechanisms along with the challenges that arise from sharing information across mechanism boundaries in Chapter 3. We then presented an architecture for sharing network context in Chapter 4, and an extensive evaluation of our design in Chapter 5.

To conclude this thesis, we now discuss our findings in relation to the goals defined in the introduction, and give an outlook on future work that can built upon our architecture.

6.1 CONCLUSION

The extensive sharing of network context across mechanisms further boosts the deployment of new and innovative services. With this, we are able to overcome the limitations of current mechanisms, and the sharing allows us to include legacy mechanisms such as routing protocols into new network architectures.

Additionally, the extensive sharing of information allows operators to fulfill even tight SLAs as it allows for a highly dynamic scaling of mechanisms such as NFs.

However, current systems are only able to share information between a very narrow set of mechanisms, often restricting the exchange of information to a single implementation. This limits the benefits gained from sharing, and continues to collect information in silo style storage.

To overcome this limitation, we presented an architecture to break these silos open and to enable the network wide sharing of context information between mechanisms on all layers of the network stack. The extensive sharing allows us to migrate flows between multiple instances of a single NF as well as seamlessly transition between routing protocols in a WMN.

We now revisit the goals of this thesis defined in Section 1.2, and show that our architecture is able to satisfy the requirements of multiple mechanisms running in a diverse set of networks while only marginally influencing the performance of the overall system.

6.1.1 *Enable Native Context Sharing Across Mechanisms*

The core enabler for extensive sharing of network context is the strict separation of functionality and state within each mechanism along with an API to store the information in an external context management system. This separation ensures that all relevant information is made available to other mechanisms that are also connected to this context management system, and the mechanism is able to benefit from previously contributed information.

On the one hand, we presented STEAN—a Storage and Transformation Engine for Advanced Networking context—that provides a generalized context storage, and offers a simple XML-base API to access this storage. The STEAN architecture supports any type

of context and is thus able to collect and disseminate the information of all connected mechanism, regardless of their individual requirements.

The evaluation shows that we are able to support fundamentally different mechanisms such as NFs and WMN routing protocols using the same implementation, only adjusting the context model—the base context—to the specific requirements.

Additionally, we show that STEAN only imposes a minimal performance overhead, and in some use cases is even able to provide faster migration times than other state of the art systems.

On the other hand, we introduced a blueprint for designing transition aware mechanisms that allows us to 1. speed up the development cycle of new mechanisms by providing crucial elements that are required for context sharing, and 2. quickly adapt existing mechanisms to become sharing-enabled by using our lightweight client side framework to integrate the required functionality.

We show how mechanisms that are based on different operational principles—namely a reactive and a proactive routing protocol—can be implemented within our framework and how these mechanisms use common elements to enable context sharing between them.

We have presented the two main components for sharing context information between a large set of different mechanisms. Our proposal enables mechanism designers to include context sharing as a first order principle into their architecture, and allows developers to easily include the sharing paradigm into new as well as existing mechanisms.

6.1.2 *Enable the Transformation of Context Information*

Transformations are a major enabler for this extensive sharing as they allow us to support a large variety of mechanisms without the need to adapt the internal state of these systems. The support for transformations by the context management system does not force developers to completely redesign their mechanism but enables them to include context sharing with only minimal changes to existing implementations.

STEAN supports transformations by design and the architecture is centered around this core feature. The transformation functions act as an additional layer between the mechanism and the storage backend, and allow STEAN to receive information “as is” and provide context “as needed”. This layer provides mechanism designers with the ability to built their own transformation functions, and customize the behavior of STEAN to the requirements of the mechanism without loosing the generality of the context storage.

We not only provided an architectural view of transformations in this thesis, but also presented guidelines for designing and implementing those functions to reduce the computational overhead and ensure the feasibility of our approach even under high load. This includes that transformations need to be developed in close interaction with the design of the base context used by the STEAN storage component to reduce the complexity of the required transformations whenever possible.

Additionally, transformations should not be used to implement features of the mechanism but only to execute functionality necessary for sharing context. While the former is technically possible, it would weaken the strict separation of functionality and context and thus reduce the generality of our architecture.

Our evaluation shows that transformation functions—when designed according to our guidelines—only have a minimal impact on the performance of STEAN.

In conclusion, the transformation functions provided by STEAN enable us to not only share state between instances of the same implementation but to extend the sharing of networking context beyond these boundaries.

6.1.3 *Enable Transition Support in Mechanisms*

In order to efficiently reconfigure the network and gain the maximum benefits from the extensive sharing of context information, it is necessary to provide mechanisms that are built with support for transitions as a first order principle. This allows us to efficiently exchange mechanisms with the same high level functionality such as routing but that are based on different a different set of requirements to best accomodate the current network situation. Along with an orchestration infrastructure that is able to efficiently and reliably execute those transitions, we can provide a more dynamic and flexible network management.

In this thesis, we presented a blueprint for such a switching architecture that uses a modular design to compose the mechanisms, and enables us to closely interact with each component. The design not only includes elements for creating the actual mechanism functionality but also for providing transition support. This tight integration between the mechanism and the transition system allows for a fine granular control over the mechanism features such as timers, and enables us to reduce the overhead introduced by a transition.

Our implementation is based on the Click Modular Router and provides dedicated elements for controlling the data path within the implementation as well as a standardized API for interacting with the included mechanisms. The controller infrastructure used for coordinating the mechanism switch across multiple nodes is implemented around a simple text based API that allows us to exchange individual controller elements with other implementations to provide a more advanced feature set such as distributed switching decisions and coordination.

The evaluation of our approach shows that we are able to dynamically switch between mechanisms during runtime without any overhead during normal operations, and that seamless transitions can be achieved with the support of a context management system such as STEAN.

We are able to natively support transitions within mechanisms by providing a modular architecture for a switching system that is not only able to select the currently active mechanism by redirecting the data flow but also interact with the internal functionality of each mechanism in order to reduce the operational overhead.

6.2 OUTLOOK

The future work on context sharing in networks is twofold. The core aspects are 1. the extension of the context management system, and 2. further work on the design and implementation of transition aware mechanisms. In the following, we provide an outlook on those issues and point to possible directions of research.

6.2.1 *Cooperation of Multiple STEAN Instances*

In this thesis, we assumed a single instance of STEAN that is responsible for collecting and disseminating the context information from all mechanisms within a functional area of the network, e.g., a single host, a group of NFs or a network segment.

On the one hand, this allowed us to share the context information available within the functional area across all mechanisms, on the other hand it also limited the sharing to a small subset of the complete network.

The federation of multiple STEAN instances can overcome this limitation, and enables the sharing of context between multiple functional areas of a network without sacrificing the principle of keep information locally close to the mechanisms. The integration of multiple instances of a context management system requires an extension that is able to decide if the requested context information is available locally—and can thus be served directly—or if the data has to be requested from another instance.

Additionally, we introduced a single point of failure by only running a single instance of STEAN that is responsible for managing the context information of multiple mechanisms or even failover groups.

This drawback could be overcome by introducing a distributed storage backend across multiple instances where each piece of context information is stored in multiple locations. The distribution of context storage would require the design of an alternative storage component that is able to dynamically balance the context information between instances depending on the current request volume, to 1. reduce the storage load on the system, and 2. keep the required information close to the end system.

The current STEAN architecture already allows for a replacement of the storage backend, as the design is based on distinct components and all components are only loosely coupled.

6.2.2 *Performance Improvements*

Our evaluation has shown that the design and implementation of STEAN is capable of handling thousands of requests per second without any significant delay in mechanism operations.

However, we also identified that the current usage of sockets to communicate between mechanisms and the STEAN instance is a major performance bottleneck. The socket communication is responsible for 45 % of the local forwarding time on a WMN node when running routing protocols that are backed by STEAN.

Future directions of research could include the usage of directly shared memory space. This can be implemented locally using Direct Memory Access (DMA) if the mechanisms run on the same node as the context management system, or by leveraging systems like RAMCloud [54].

This also requires the redesign of transformation functions as they need to be directly executed on the memory contents rather than acting as an additional layer between mechanism and context storage.

6.2.3 *Generation of Transformation Functions*

Currently, transformation functions that map the mechanism specific context to the base context provided by STEAN need to be provided by the mechanism developers as the design and implementation of transformations requires specific knowledge about the internal operations and the requirements of the mechanism. Reducing the overhead of creating specialized transformations for each mechanism would increase the adaptation of context management systems as one of the main drawbacks for developers would be removed.

This could be achieved by using static code analysis on the data structures used to handle context information within the mechanism, and by applying dynamic analysis of memory structures along with the mechanisms output.

The results of this analysis can then either be mapped to an existing base context—in case a new mechanism needs to be integrated into an existing environment—or the results of multiple analysis are federated to find the optimal base context for a given combination of mechanisms.

6.2.4 *Anticipate Mechanism Behavior*

STEAN provides a management solution for network context as defined in Section 2.1, including historical records and monitoring data. This information can be leveraged to gain insights on the mechanism behavior and develop strategies to transition between those mechanisms. We could—assumed that the information was already gathered over time—learn on what mechanism performs best during which conditions, and which transitions were successful, i.e., improved the overall network performance.

Therefore, we propose the in-depth analysis of context information, and the possible application of machine learning strategies as a direction for future research. This would allow us to gain a deep understanding on how mechanism properties influence the network behavior under certain environmental conditions, and enable operators to better react on those conditions.

Moreover, the systematic collection and evaluation of the gathered information would allow us to anticipate possible bottlenecks and execute appropriate transitions even before network problems arise.

6.2.5 *Complex Transitions*

This thesis focused on transitions where only one mechanism class—such as routing protocols or type of NF—is switched at a time.

However, during practical operation, it might be necessary to switch multiple mechanisms either directly after each other or even at the same time, e.g., when an application layer mechanisms depends on the properties of a specific transport protocol, or a secure routing protocol requires a certain cryptographic scheme.

Those complex transitions where dependencies between mechanisms exist and a transition between operational states require the exchange of multiple mechanisms are areas that require further investigation.

One research question that might arise is how to model and resolve those dependencies to ensure that the network is always in an operational state, and a possible resolution

could be the necessity to execute transitions using intermediate mechanisms on one layer that are chosen to allow the seamless switch on another layer. The intermediate mechanisms are not run for a longer period but only ensure the required properties are provided while the transition is in progress.

6.2.6 *Transitions of Finer Granularity*

While our mechanism architecture presented in Section 4.2 is modular—thus allowing for switches on sub-mechanism level—in this work we have focused on transitions between mechanisms as a whole.

Possible future work in this area is the evaluation on how mechanisms behave if core elements such as queueing strategies in routing protocols or congestion control algorithms in transport protocols are exchanged during runtime.

This would allow for lightweight network stacks that include multiple sub-mechanisms, and thus enable transitions on very low powered hardware such as sensor nodes or Internet of Things (IoT) devices.

6.2.7 *Decentralized Ochestration*

In our experiments, we simplified the control of the transition to be centralized and manually executed using an out-of-band channel. This allowed us to tightly control the network during a switch, and easily recover from any failure during the transition.

In practical deployments, however, such a central instance that is able to orchestrate the transition might not exist. Therefore, it is necessary to investigate decentralized schemes that use in-band messaging to coordinate the switch.

Moreover, a suitable decision metric which signifies whether to switch or not and which transitions to perform, needs to be identified. This decision ideally is also made in a distributed fashion such as using appropriate voting schemes where all nodes participating in the network decide on the transition depending on their local state and the available context information.

ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation (DFG) as part of the Collaborative Research Center (CRC) 1053 “MAKI—Multi-Mechanisms Adaptation for the Future Internet”.

BIBLIOGRAPHY

[1] M. Adeyeye and P. Gardner-Stephen. The Village Telco project: a reliable and practical wireless mesh telephony infrastructure. *EURASIP Journal on Wireless Communications and Networking (JWCN)*, 2011(1), 2011.

[2] C. Adjih, T. Clausen, A. Laouiti, P. Mühlethaler, and D. Raffo. Securing the OLSR protocol. In *Proceedings of the 2nd IFIP Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, 2003.

[3] M. Afanasyev, T. Chen, G. M. Voelker, and A. C. Snoeren. Analysis of a mixed-use urban wifi network: When metropolitan becomes neapolitan. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2008.

[4] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger. Anatomy of a Large European IXP. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[5] M. Agrawal, S. R. Bailey, A. Greenberg, J. Pastor, P. Sebos, S. Seshan, K. van der Merwe, and J. Yates. RouterFarm: Towards a Dynamic, Manageable Network Edge. In *Proceedings of the 2006 SIGCOMM Workshop on Internet Network Management (INM)*, 2006.

[6] Amazon Web Services, Inc. Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/.

[7] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of the 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2012.

[8] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P.R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems (TODS)*, 1(2), 1976.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[10] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[11] M. Bernaschi, F. Casadei, and P. Tassotti. SockMi: a solution for migrating TCP/IP connections. In *Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2007.

[12] A. Botta, A. Dainotti, and A. Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks"*, 56(15), 2012.

[13] R. Braden. RFC 1122: Requirements for Internet Hosts – Communication Layers. Technical report, The Internet Engineering Task Force (IETF), 1989.

[14] R. Braden. RFC 1123: Requirements for Internet Hosts – Application and Support. Technical report, The Internet Engineering Task Force (IETF), 1989.

[15] R. Bush and D. Meyer. RFC 3439: Some Internet Architectural Guidelines and Philosophy. Technical report, The Internet Engineering Task Force (IETF), 2002.

[16] V. G. Cerf and E. Cain. The DoD internet architecture model. *Computer Networks*, 7(5), 1983.

[17] Cisco Systems Inc. Graceful Restart, Non Stop Routing and IGP routing protocol timer Manipulation. http://bit.ly/1NgpVJO.

[18] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A Knowledge Plane for the Internet. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2003.

[19] M. Conti, G. Maselli, G. Turi, and S. Giordano. Cross-layering in mobile ad hoc network design. *Computer*, 37(2), 2004.

[20] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[21] DE-CIX Management GmbH. DE-CIX – German Internet Exchange. https://www.de-cix.net.

[22] P. Dutta, S. Jaiswal, D. Panigrahi, K.V. M. Naidu, R. Rastogi, and A. Todimala. VillageNet: A low-cost, 802.11-based mesh network for rural regions. In *Proceedings of the 2nd International Conference on Communication Systems Software and Middleware (COMSWARE)*, 2007.

[23] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe. The Case for Separating Routing from Routers. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, 2004.

[24] E. Fjellskål. Passive Real-time Asset Detection System. http://gamelinux.github.io/prads/.

[25] A. Frömmgen, M. Hassan, R. Kluge, M. Mousavi, M. Mühlhäuser, S. Müller, M. Schnee, M. Stein, and M. Weckesser. Mechanism Transitions: A New Paradigm for a Highly Adaptive Internet. Technical report, Technische Universität Darmstadt, 2016.

[26] A. Gember, R. Grandl, J. Khalid, and A. Akella. Design and Implementation of a Framework for Software-defined Middlebox Networking. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2013.

[27] A. Gember-Jacobson and A. Akella. Improving the Safety, Scalability, and Efficiency of Network Function State Transfers. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[28] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2014.

[29] M. Guerrero-Zapata. Secure Ad hoc On-Demand Distance Vector (SAODV) Routing. Technical report, Technical University of Catalonia (UPC), 2006.

[30] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. SDX: A Software Defined Internet Exchange. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2014.

[31] T. Hossmann, F. Legendre, P. Carta, P. Gunningberg, and C. Rohner. Twitter in Disaster Mode: Opportunistic Communication and Distribution of Sensor Data in Emergencies. In *Proceedings of the 3rd Extreme Conference on Communication: The Amazon Expedition (ExtremeCom)*, 2011.

[32] P. Jacquet and T. Clausen. RFC 3626: Optimized Link State Routing Protocol (OLSR). Technical report, The Internet Engineering Task Force (IETF), 2003.

[33] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2013.

[34] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless Network Functions. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[35] M. Kalicinski. RapidXML. http://rapidxml.sourceforge.net/.

[36] M. J. Karam and F. A. Tobagi. Analysis of the Delay and Jitter of Voice Traffic Over the Internet. In *Proceedings of the 20th Conference on Computer Communications (INFOCOM)*, 2001.

[37] E. Keller, J. Rexford, and J. E. van der Merwe. Seamless BGP Migration with Router Grafting. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[38] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[39] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3), 2000.

[40] B. Kothandaraman, M. Du, and P. Sköldström. Centrally Controlled Distributed VNF State Management. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[41] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1), 2015.

[42] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet Routing Convergence. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2000.

[43] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[44] M. Lindeberg, V. Goebel, and T. Plagemann. CLiSuite: Simplifying the Development of Cross-layer Adaptive Applications. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing (MW4NG)*, 2012.

[45] J. Lloyd. Botan Cryptographic Library. https://botan.randombit.net.

[46] J. R. Lorch, A. Baumann, L. Glendenning, D. T. Meyer, and A. Warfield. Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[47] K. Lougheed and Y. Rekhter. RFC 1105: A Border Gateway Protocol (BGP). Technical report, The Internet Engineering Task Force (IETF), 1989.

[48] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*, 2009.

[49] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2), 2008.

[50] Nagios Enterprises. Nagios – The Industry Standard In IT Infrastructure Monitoring. https://www.nagios.com.

[51] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[52] H. Nishiyama, M. Ito, and N. Kato. Relay-by-smartphone: realizing multihop device-to-device communications. *IEEE Communications Magazine*, 52(4), 2014.

[53] A. Osseiran, V. Braun, T. Hidekazu, P. Marsch, H. Schotten, H. Tullberg, M. A. Uusitalo, and M. Schellman. The Foundation of the Mobile and Wireless Communications System for 2020 and Beyond: Challenges, Enablers and Technology Solutions. In *Proceedings of the 77th IEEE Vehicular Technology Conference (VTC Spring)*, 2013.

[54] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4), 2010.

[55] C. Perkins, E. Belding-Royer, and S. Das. RFC 3561: Ad hoc On-Demand Distance Vector (AODV) Routing. Technical report, The Internet Engineering Task Force (IETF), 2003.

[56] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC)*, 2013.

[57] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[58] K. N. Ramachandran, E. M. Belding-Royer, and K. C. AImeroth. DAMON: a distributed architecture for monitoring multi-hop mobile networks. In *Proceedings of the 1st Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, 2004.

[59] Y. Rekhter, T. Li, and S. Hares. RFC 4271: A Border Gateway Protocol 4 (BGP-4). Technical report, The Internet Engineering Task Force (IETF), 2006.

[60] N. Richerzhagen, D. Stingl, B. Richerzhagen, A. Mauthe, and R. Steinmetz. Adaptive Monitoring for Mobile Networks in Challenging Environments. In *Proceedings of the 24th International Conference on Computer Communication and Networks (ICCCN)*, 2015.

[61] P. Richter, N. Chatzis, G. Smaragdakis, A. Feldmann, and W. Willinger. Distilling the Internet's Application Mix from Packet-Sampled Traffic. In *Proceedings of the 16th International Conference on Passive and Active Measurement (PAM)*, 2015.

[62] E. Rosen and Y. Rekhter. RFC 4364: BGP/MPLS IP Virtual Private Networks (VPNs). Technical report, The Internet Engineering Task Force (IETF), 2006.

[63] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[64] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proceedings of the 10th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2011.

[65] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[66] V. Srivastava and M. Motani. Cross-layer design: a survey and the road ahead. *IEEE Communications Magazine*, 43(12), 2005.

[67] H. V. Styn. Tcpdump Fu. *Linux journal*, 2011(210), 2011.

[68] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A Network-state Management Service. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2014.

[69] The Icinga Project. Icinga – Open Source Monitoring. https://www.icinga.org.

[70] V. Valancius and N. Feamster. Multiplexing BGP Sessions with BGP-Mux. In *Proceedings of the 3rd International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2007.

[71] S. Vissicchio, L. Vanbever, C. Pelsser, L. Cittadini, P. Francois, and O. Bonaventure. Improving Network Agility With Seamless BGP Reconfigurations. *IEEE/ACM Transactions on Networking (TON)*, 21(3), 2013.

[72] L. Wang, M. Saranu, J. M. Gottlieb, and D. Pei. Understanding BGP Session Failures in a Large ISP. In *Proceedings of the 26th Conference on Computer Communications (INFOCOM)*, 2007.

[73] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration As a Network-management Primitive. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2008.

[74] M. Werner, J. Kaiser, M. Hollick, E. Weingärtner, and K. Wehrle. A Blueprint for Switching Between Secure Routing Protocols in Wireless Multihop Networks. In *Proceedings of the 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, 2013.

[75] M. Werner, J. Schwandke, M. Hollick, O. Hohlfeld, T. Zimmermann, and K. Wehrle. STEAN: A Storage and Transformation Engine for Advanced Networking context. In *Proceedings of the IFIP Networking Conference (IFIP Networking)*, 2016.

[76] R. Winter, J. H. Schiller, N. Nikaein, and C. Bonnet. CrossTalk: cross-layer decision support based on global knowledge. *IEEE Communications Magazine*, 44(1), 2006.

[77] H. Zimmermann. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4), 1980.

All web pages cited in this work have been checked in September 2017. However, due to the dynamic nature of the World Wide Web, their long-term availability cannot be guaranteed.

A

CHARACTERIZING THE TRAFFIC GAP

One of the main challenges when introducing a mechanism switch during runtime is the reestablishment of context information as described in Section 3.4.1. The transition will lead to a gap in data traffic until all communication paths and context information are established and the network is fully operational again.

We discuss additional parameters and metrics—not or only partially reviewed in Section 5.2—that influence the duration of the traffic gap along with metrics that help us to analyze the network traffic for gap times during a switch and to identify the contributing factors.

A.1 PARAMETERS

Both routing protocols under test in Section 5.2 have several parameters that determine the performance of the network. We distinguish between traffic independent and traffic dependent parameters that either generate additional load by periodically emitting packets or significantly influencing the forwarding performance of the network.

A.1.1 *OLSR*

A.1.1.1 *Traffic Independent*

The *Link Hysteresis* allows the protocol to decide when a wireless link should be considered valid. It helps to avoid the phenomena of "on-off-neighbors" by introducing a pending link state in which the wireless link to the neighbor is considered to be not established even if some HELLO messages are received. The hysteresis has an upper bound and a lower bound value. As we run only static nodes in our network and thus the characteristics of the wireless link remain the same, we chose to keep the link hysteresis fixed.

The *Willingness* of a node describes how this node is disposed to forward traffic on behalf of other nodes. This parameter influences the routing decisions and might be based on external factors such as the availability of a permanent power supply or computational resources. We ran all our experiments in a homogeneous setup and therefore decided to set the willingness of all nodes to the same value to not prefer one of the nodes over another.

A.1.1.2 *Traffic Dependent*

The *HELLO Interval* defines the emitting interval of link sensing announcements and influences the discovery of new or lost neighbors. Additionally, HELLO messages are the most frequent control messages and thus can account for a high additional load. Nodes first need to discover at least one neighbor before they are reachable from other members of the network. Therefore, we expect a high influence of the HELLO Interval on the gap time as HELLO messages might get delayed or even lost during the initialization phase

and the neighbor discovery then takes a large amount of time. In our experiments, we used three different intervals to account for different networking situations where nodes might move fast to an almost static network.

Directly related to the emitting interval is the *Neighbor Hold Time* which describes the time, an entry in the neighbor list is considered valid even if no additional HELLO messages are received. We have chosen three different factors of the HELLO interval for the Neighbor Hold Time to account for different environmental conditions as well as load pattern.

As OLSR is a proactive protocol, the actual routing information is transmitted using Topology Control (TC) messages. Therefore, the *TC Interval* influences the built up time of the routing tables once neighbors are discovered. These messages are flooded through the whole network and can account for a high additional load.

The *TC Hold* time defines how long an entry in the routing table should be kept without any refresh. Once this time expires and no TC message with the according information was received, the entry is removed and the path is not used any longer.

For an overview of possible values for each parameter we refer to Table A.1.

PARAMETER	VALUES
HELLO_INTERVAL	1000 ms, 2000 ms*, 5000 ms, 10000 ms
TC_INTERVAL	4000 ms, 5000 ms*, 15000 ms, 20000 ms
NEIGHBOR_HOLD	2 × HELLO_INTERVAL, 3 × HELLO_INTERVAL*, 4 × HELLO_INTERVAL, 8 × HELLO_INTERVAL
TC_HOLD	2 × TC_INTERVAL, 3 × TC_INTERVAL*, 4 × TC_INTERVAL, 8 × TC_INTERVAL

Table A.1: OLSR parameters. A * denotes the default parameter setting from the RFC.

A.1.2 *AODV*

A.1.2.1 *Traffic Independent*

The *Network Diameter* describes the maximum number of hops a route in the network can consist of. We are running a small and dense network and therefore lowered the net diameter to avoid long running expanding ring searches when no route is available. All experiments were executed in a static setup which allowed us to keep the parameter static.

We used a high *Node Traversal Time* to account for the low computational performance of our mesh nodes. This parameter depicts the maximum retention time on a single node and even though our experiments did not show any performance bottlenecks on the CPU, the messages might still need a significant time to be forwarded.

The *Route Request Retry* count is set to a static value across all tests but high enough that when the value is reached, the running experiment is also finished.

Using the *Active Route Timeout* parameter, the protocol removes unused routes after a certain period. We are constantly transmitting data over all routes and therefore this threshold never gets reached and can remain static over all tests.

A.1.2.2 *Traffic Dependent*

AODV is a reactive ad-hoc routing protocol and therefore only has very few messages that are sent periodically.

The *HELLO Interval* defines the emitting interval of neighbor discovery messages. These are broadcasted periodically to announce the existence of a node and its willingness to participate in the network. Each node has to at least receive one valid HELLO message before it can participate in the network and request routes for other destinations. HELLO messages are the only packets sent periodically by AODV. We have chosen three different emitting intervals to account for different node behaviors from fast moving participants to an almost static network.

Directly related to the HELLO Interval is the *Allowed HELLO Loss* which depicts the number of HELLO messages to be lost before a node should assume that the link to the respective neighbor is currently lost. In our experiments, we used three different values for the Allowed HELLO Loss to account for different load pattern as well as environmental conditions that influence the reception of broadcast messages.

The *RREQ Retries* parameter sets the number of Route Requests sent by a node before it considers a destination not reachable over the network. We kept this parameter high but static to account for a potential high loss of Route Requests during the high traffic experiments. This setting was chosen in favor of different values for the RREQ Retries parameter as it does not directly impose more traffic but the actual finding of routes highly relies on this parameter.

Table A.2 gives an overview of the parameters and sensible values for conducting experiments.

For AODV, the Route Request (RREQ) messages are expected to create substantial overhead when a protocol switch is performed. These messages are only sent on demand but broadcasted through the whole network to discover routes.

PARAMETER	VALUES
HELLO_INTERVAL	1000 ms*, 5000 ms, 10000 ms
ALLOWED_HELLO_LOSS	2*, 4, 8
RREQ_RETRIES	2*, 5

Table A.2: AODV parameters. A * denotes the default parameter setting from the RFC.

A.2 METRICS

During our experiments we record different metrics that can be used to characterize the behavior of a WMN and the gap in networking traffic when switching mechanisms in particular.

A.2.1 *Duration of the Traffic Gap*

The most important metric is the time where no data traffic in the network is forwarded. During this interval, no communication between end nodes is possible and the user experiences a degradation in service.

We define the duration of the traffic gap as the time between the reception of the switch command by a node and the first successful delivery of a data packet after the switch was initiated. During this time a routing protocol typically has to built an initial routing table that contains at least one entry for the observed flow. Thus, the protocol does not have to be in steady state but needs to offer end to end connectivity.

During the gap phase there might also be data traffic still forwarded to the next hop that is already in the outbound buffer. These packets are however not necessarily delivered to the final destination.

A.2.2 *Communication Overhead*

When a new protocol is bootstrapped, typically a large number of messages has to be exchanged by the participating nodes. The neighbors of each node have to be discovered, routes have to be selected and forwarding nodes have to be chosen. This leads to an increased overhead of control messages until the network is stable again. The number of messages is typically even larger when there are already data packets waiting to be delivered. We record the number and size of control messages along with the data traffic to calculate the overhead each mechanism produces.

A.2.3 *System Load*

The forwarding performance of intermediate nodes also depends on the load the system has to handle during operation. This load can be caused by the forwarding itself but also by the management and control overhead of the protocols.

A.3 SUMMARY

Dynamic adaptation of mechanisms is a key to optimize the performance of communication networks. In Section 5.2, we showed that a traffic gap exists when switching between different routing protocols in a WMN.

This gap does not only prevail if the load on the network is high but also on low to moderate data rates. While the gap is still small on moderate load of the network it suddenly grows rapidly once a certain throughput threshold is reached.

We also argue that the gap not only depends on the traffic load imposed but also on the parameter settings of the involved mechanisms. Our results show a high influence of the number of control messages and the emitting interval of these messages. These parameters are however not only important for the characterization of the traffic gap but also for the reliable operation of the mechanisms themselves. A dynamic adjustment of these parameters depending on the current network state as well as the operation mode might be feasible and will improve the overall network performance not only during the transition of communication protocols but also during normal operations.

B

CURRICULUM VITÆ

PERSONAL INFORMATION

Name	Marc Werner
Date of Birth	27 Febrary 1983
Place of Birth	Alzenau i. Ufr., Germany
Nationality	German

EDUCATION

since 2011	Technische Universität Darmstadt, Darmstadt, Germany Doctoral candidate at the Department of Computer Science
2003 – 2011	Technische Universität Darmstadt, Darmstadt, Germany Studies of Computer Science Degree: Master of Science
1993 – 2005	Franziskaner Gymnasium Kreuzburg, Großkrotzenburg, Germany Degree: Abitur (German high school diploma)

WORK EXPERIENCE

since 2017	Continental Teves AG & Co. oHG, Frankfurt, Germany Specialist Security & Privacy
2011 – 2017	Technische Universität Darmstadt, Darmstadt, Germany Research associate at the Secure Mobile Networking Lab

SUPERVISED THESES

M.Sc. Thesis	Raphael John. Secure Context Migration between IEEE 802.11 Wireless Networks
B.Sc. Thesis	Daniel Kratschmann. Design and Implementation of a Service-Oriented Architecture for Large-Scale Testbed Management
B.Sc. Thesis	Paul Wiedenbeck. Enabling Seamless Transitions between Cryptographically Secured Protocols in Networks
B.Sc. Thesis	Florian Herrmann. Understanding the traffic gap when switching routing protocols
M.Sc. Thesis	Tobias Lange. Secure Protocol Transitions in Mobile Ad hoc Networks
M.Sc. Thesis	Johannes Schwandke. Design, Implementation and Evaluation of a System Information Service

B.Sc. Thesis	Benedikt Rudolph. CLICK2WARP—Integrating the click modular router and the wireless open-access research platform for network-scale experiments Service
B.Sc. Thesis	Claudius Kleemann. Network ID—Self-Provisioning Service Proxy
M.Sc. Thesis	Paul Klobuszenski. Mobile Phones as Sensors for Intrusion Detection in Wireless Mesh Networks
M.Sc. Thesis	Jörg Kaiser. Secure Modular Protocols for Wireless Multi-hop Networks

Darmstadt, 11. September 2017

C

AUTHOR'S PUBLICATIONS

PRIMARY AUTHOR

- M. Werner, D. Kratschmann, M. Hollick. Panopticon: Supervising Network Testbed Resources. In *41st Conference on Local Computer Networks (IEEE LCN, Demo Track), November 2016*
- M. Werner, J. Schwandke, M. Hollick, O. Hohlfeld, T. Zimmermann, K. Wehrle. STEAN: A Storage and Transformation Engine for Advanced Networking Context. In *Proceedings of the IFIP Networking Conference (IFIP Networking), May 2016*
- M. Werner, T. Lange, M. Hollick, T. Zimmermann, K. Wehrle. Mind the Gap—Understanding the Traffic Gap when Switching Communication Protocols. In *Proceedings of the 1st KuVS Workshop on Anticipatory Networks, September 2014*
- M. Werner, J. Kaiser, M. Hollick, E. Weingärtner, K. Wehrle. A Blueprint for Switching Between Secure Routing Protocols in Wireless Multihop Networks. In *Proceedings of the 14th International Symposium on a World of Wireless, Mobile and Multimedia Networks (IEEE WoWMoM, D-SPAN Workshop), June 2013*

CO-AUTHOR

- R. do Carmo, M. Werner, P. Klobuszenski, M. Hollick. Not the Enemy but the Ally: Increasing the Security of Wireless Networks Using Smartphones (extended abstract). In *7. Essener Workshop "Neue Herausforderungen in der Netzsicherheit" (EWNS), April 2013*
- R. do Carmo, M. Werner, M. Hollick. Signs of a Bad Neighborhood: A Lightweight Metric for Anomaly Detection in Mobile Ad Hoc Networks. In *Proceedings of the 8th ACM International Symposium on QoS and Security for Wireless and Mobile Networks (ACM Q2SWinet), October 2012*
- A. Reinhardt, P. Baumann, D. Burgstahler, M. Hollick, H. Chonov, M. Werner, R. Steinmetz. On the Accuracy of Appliance Identification Based on Distributed Load Metering Data. In *Proceedings of the 2nd IFIP Conference on Sustainable Internet and ICT for Sustainability (SustainIT), October 2012*

D

ERKLÄRUNG LAUT §9 DER PROMOTIONSORDNUNG

Ich versichere hiermit, dass ich die vorliegende Dissertation selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmitteln verfasst habe. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch nicht zu Prüfungszwecken gedient.

Darmstadt, 11. September 2017

Marc Werner