STEAN: A Storage and Transformation Engine for Advanced Networking Context

Marc Werner*, Johannes Schwandke*, Matthias Hollick*,
Oliver Hohlfeld†, Torsten Zimmermann† and Klaus Wehrle†
* Technische Universität Darmstadt, Secure Mobile Networking Lab (SEEMOO)
† RWTH Aachen University, Chair of Communication and Distributed Systems (COMSYS)

***Abstract*—Legacy Internet 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 Networking (SDN) approaches in unison with Network Functions Virtualization (NFV) allow for more flexibility, yet they are currently restricted to a limited set of state migration options. Impeding the sharing of networking and system state severely limits the ability to optimally manage resources and dynamically adapt to a desirable overall configuration. We propose a generalized way to collect, store, *transform*, and share context between NFs in both the legacy Internet and NFV/SDN-driven systems. To this end, 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. By means of experimentation, we show that STEAN covers a diverse set of challenging use cases in legacy systems as well as in NFV/SDN-enabled systems.**

I. Introduction

Network management currently undergoes massive changes towards realizing a more flexible management of complex networks. Recent efforts include 1) rethinking the control plane design by applying operating system design principles to realize Software Defined Networking (SDN), and 2) Network Function Virtualization (NFV) inspired by the success of virtualization in the server market. These advances aim at a more flexible and dynamic service deployment, increased resource utilization, improved energy efficiency, vendor independence, and ,ultimately, decreased operational costs.

While these techniques advance *packet processing* and *service control*, they do not address *state management*. However, to achieve the true benefits of network and service virtualization as well as control plane programmability, scalable state sharing is of high importance—in particular when attempting to virtualize stateful network functions (NFs). This requirement has lead to the development of various systems that allow explicit state migration between NFs such as Split/Merge [1], OpenNF [2] or StatelessNF [3].

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 Fig. 1 (a).

We advocate to *break these silos open* and allow state to be shared within the entire ecosystem of a network, ranging from

ISBN 978-3-901882-83-8

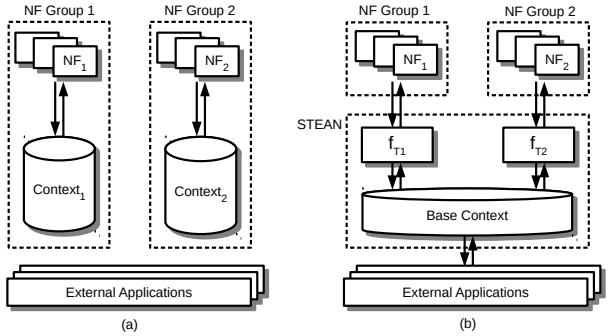

Fig. 1. Current state sharing frameworks only allow sharing between functions of the same group (a). We enable sharing context between different groups by using a common base context and mapping the function specific context using the transformation functions f_{T1} and f_{T2} (b).

SDN controllers over NFs to routers and protocol implementations (Fig. 1 (b)). By following this approach, we pave the way for realizing a *network state plane* in which network state is decoupled from the implementation of NFs similar to the decoupling of the control and data plane introduced by SDN. This decoupling of packet processing and network state will lead to a more flexible and dynamic network management, and further boost the deployment of new and innovative services.

New network management functions can use the state plane to easily aggregate state of multiple functions operating at different layers without requiring explicit support in each function. The proposed state decoupling thus enables new ways for network state cross-layering that is harder to achieve in current isolated solutions. In a similar fashion, network state can be migrated between functions more easily without explicit support.

As state migration between NFs is challenged by different state representations, we further advocate the use of *state transformation functions*. Transformation functions allow us to leave the internal state representation of the legacy systems unaltered while still sharing state with other systems. Since the transformation functions operate within the state plane rather than in each network function, we limit the explicit need for supporting state migrations. This approach surpasses the flexibility of existing solutions and enables us to generalize state management across multiple systems.

Extending the sharing and re-use of information between systems and protocols *beyond* state information further opens networks to a more flexible and dynamic management. We call

this extended set of information the *context* of a networked system. The context includes the internal state as well as the metadata describing how to interpret the stored information. Additionally, the context includes the current configuration parameters, monitoring information and historical records.

We summarize our contributions as follows:

1) We propose a *Storage and Transformation Engine for Advanced Networking Context (STEAN)*. It provides a generalized way to share context in a diverse set of core functionality such as routing, network processing, and dynamic protocol adaptation. This relies on collecting and managing context from different sources (e.g., NFs, 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 systems and protocols, and stores and persists the current context to re-use at later points in time.

2) We introduce transformation functions that allow for context sharing between systems that were not originally built with sharing in mind. Transformations allow STEAN to be integrated into legacy systems and to interoperate with arbitrary protocols, which permits the seamless extension of existing protocol 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.

3) We demonstrate the functionality of STEAN and evaluate its general applicability as well as its performance in two selected use cases.

II. Use Cases

We provide several motivating use cases that show why broader sharing of context is beneficial for NFs as well as for existing systems such as routing protocols in Wireless Multihop Networks (WMNs). Moreover, we show that explicit support of context sharing is essential for the development of future networks. To demonstrate the general applicability, we discuss both abstract and specific use cases.

A. UC1. Migration of Network Functions: The migration or sharing of context is a core enabler of employing virtualized NFs. This allows NFs to not only scale dynamically but keep per-flow information consistent across all instances. For example, an IDS keeps state about each flow, and rerouting the flow to a different IDS instance can significantly impact the accuracy due to missing context. Thus, context sharing improves the detection rate while still supporting dynamic scaling and flow redirection.

B. UC2. Reconfigure Network Functions: NFs are currently unable to directly share context with other systems. All information exchanged between groups of NFs flows via a central controller, limiting the available context to a predefined set that is known to the providing and consuming NF as well as to the controller. An asset management system like PRADS [4] might want to share information about hosts and services 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.

Direct sharing of context adds robustness as it enables a decentralized context management, and avoids bottlenecks on the control plane when the shared context is large.

C. UC3. Switching Routing Protocols: Wireless Multihop Networks are a key technology in fifth generation (5G) wireless networks [5]. Today WMNs are deployed in environments where wired infrastructure is either not available or too expensive to deploy [6]. Currently, a variety of parameters and environmental conditions have to be considered when planning and deploying a WMN. These considerations determine a choice of technology and protocols 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 system configuration.

The dynamic adaptation of routing protocols provides an exit route to this dilemma. The protocol change within a WMN, however, must be seamless, without interruption of end-to-end connectivity and transparent to the end user.

III. 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 systems or protocols.

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 Network Address Translator (NAT), an SDN-enabled switch that balances the traffic load between two firewalls, and an IDS (Fig. 2). 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 different NFs operate independently of each other.

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.

A. 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

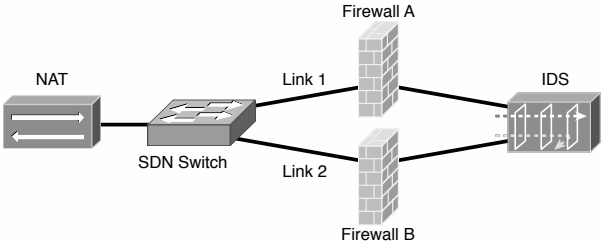

Fig. 2. Simple network to show the advantages of sharing context between different NFs. All NFs as well as the SDN controller are connected to a central STEAN instance (omitted for clarity).

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:

1) *Filter Functions* are applied during data retrieval and limit the results to the context information that is relevant to the client. 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 protocol 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. Aggregations can thus be compared to the JOIN operation 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 state the SDN rules are generated.

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 state information using transformation functions to convert the context but has to gather the missing information using system or protocol specific mechanisms. 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 protocols, 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.

B. Features

STEAN allows clients to specify the transformation functions between their context and the base context upon connection. Those functions are then called each time data 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 firewall systems—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.

C. 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 1-hop neighborhood to a 2-hop neighborhood is only partially possible, as the missing state needs to be inferred by the protocol itself.

D. 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 might reduce 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 the 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 system or protocol 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.

IV. System Design

STEAN is designed as a node-local system that manages all context information of connected clients. A *node* is not limited to a 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 Intrusion Detection Systems (UC1), or a single wireless device that is forwarding traffic in a WMN (UC3). A certain number of clients thus form a node. The design is centered around the transformation functions as the enabler for a generalized context management system.

A. Components

STEAN consists of five core components which are assigned specific tasks within our architecture and can be exchanged with other implementations. Fig. 3 gives an overview of the components and their interaction.

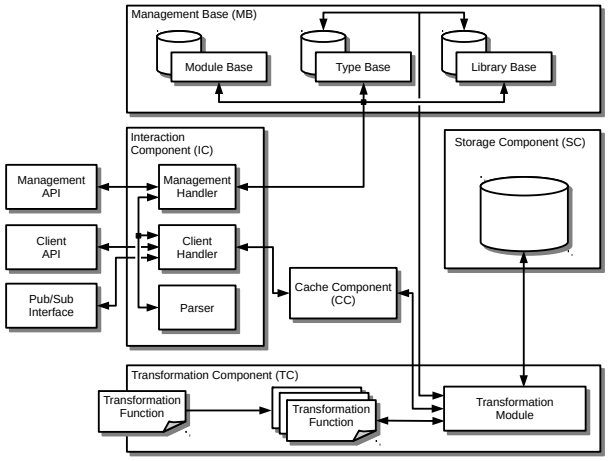

Fig. 3. Architectural overview of STEAN. The arrows show the interaction between components.

1) 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.

2) 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.

3) Transformation Component: The Transformation Component (TC) implements transformation functions as described in Section III. 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).

4) Cache Component: To be better suitable for performance critical NFs, STEAN makes intensive use of caching. The Cache Component (CC) is placed between IC and TC. The cache 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 reuse 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 reuse of cached entries as each client specifies its own context. A shared cache instead extend the number of entries per cache set and thus lead 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 reuse of cached information but the gains are much lower since transformations have to be applied to each returned result.

The cache allows to reduce the retrieval costs for state lookups, which is relevant for performance critical NFs (e.g., functions performing per-packet lookups at line rate). However, such high-speed NFs are out the of scope of this paper, and we leave their evaluation for future work.

5) 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.

B. Communication and Interaction

STEAN provides two interfaces for outside communication. The Client API is used by the accessing systems and protocols 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.

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 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.

STEAN also offers a publish-subscribe interface that notifies connected protocols when changes to subscribed annotations occur. This interface can notify connected clients such as monitoring systems when the stored context changes.

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.

V. Implementation

STEAN is implemented in C++ and runs as an independent service on the host system. We successfully tested the functionality on Linux, FreeBSD and Apple OS X.

A. Storage System

The storage system is implemented on top of a XML database using RapidXML [7], 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. 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.

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.

B. Function Libraries

Transformations are implemented as Unix shared objects 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.

C. 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.

1) UC1. Migration of Network Functions: The PRADS asset monitor [4] is a passive network monitor that allows to map the services running in a network and detect changes in real time. It uses TCP and UDP fingerprinting to identify operating systems 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.

The STEAN support for PRADS is built on top of the OpenNF [2] 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 PRADS instances using STEAN. We therefore modified PRADS to be a STEAN client while still supporting the OpenNF controller messages to initiate the migration of flows.

The PRADS instances share the complete internal state of all observed flows using STEAN. Beside a unique identifier per flow, the protocol 5-tuple and the IP protocol version, the flow state also includes 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.

As 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.

2) UC3. Switching Routing Protocols: We modified implementations of the Ad hoc On-Demand Distance Vector (AODV) routing protocol [8] as well as the Optimized Link State Routing (OLSR) protocol [9] to support UC3. The protocols are implemented using the Click Modular Router [10] framework and we extended the state handling elements to connect to STEAN. Each protocol uses a special Click element that is responsible for specifying the protocol context, and registering annotations and transformation functions during system startup. This element also handles the communication with STEAN during protocol operation.

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 protocols do not hold any local context but solely access information stored in STEAN.

We have implemented transformation functions for both routing protocols that 1) filter entries in the routing table to select only the specific route for a single packet and 2) map the format of an entry to match the internal format of the accessing routing protocol.

VI. Evaluation

We evaluate STEAN in the use cases UC1 and UC3 from Section II since they represent the diversity of possible operation scenarios for a context management system. Before we present the results from the use case study, we show the general applicability of STEAN and how the usage of transformation functions influences the system behavior.

A. General applicability: Our goal is to understand the performance of basic STEAN operations. We evaluate the behavior of STEAN using a simple client that is able to store and retrieve context information. The client inserts and reads IPv4 addresses that are either represented as a string with dots separating the octets or each octet represented as an 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.

Fig. 4 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.

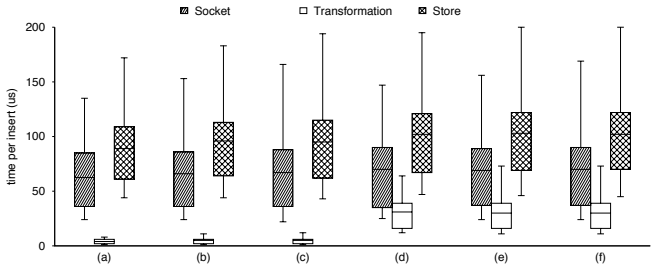

Fig. 4. Time per insert without calling a transformation function (a–c), and with calling a transformation function to convert the representation (d–f).

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 communicatin. 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 Fig. 4, we remark that additional clients have a negligible performance impact and only increase the variability of the insert time (not shown).

Fig. 5 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.

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

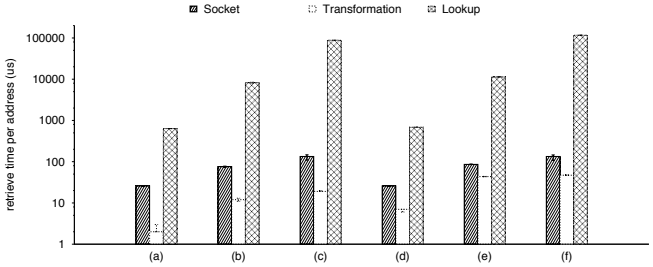

Fig. 5. Time to retrieve an address without calling a transformation function (a–c), and with calling a transformation function (d–f). The boxes represent the median and the error bars show the first and third quartile.

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.

B. UC1. Migration of Network Functions: Our goal is to compare context sharing using STEAN with current state migration systems for virtualized NFs such as OpenNF. 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 V-C1. All experiments were conducted inside a Mininet [11] instance.

The data network consists of two PRADS instances ($PRADS_1$ and $PRADS_2$) that are connected to an Open vSwitch, and a dedicated host in the data network replays a university-to-cloud trace. The trace has an overall duration of approx. 20 h and contains 70 k TCP flows, $2/3$ of which are HTTP(S) flows. On average, a flow has a duration of 35 s and 33.6 flows are active in parallel. For 13 % of the time, more than 100 flows are active in parallel.

The PRADS instances are connected to STEAN using a dedicated management network that also hosts the NF controller. The controller is responsible for initiating the migration of flows between the two PRADS instances and for reconfiguring the SDN switch during migration. The setup is depicted in Fig. 6. The experiments are run on a single machine with a Quad-core Intel Xeon CPU and 16 GB of RAM.

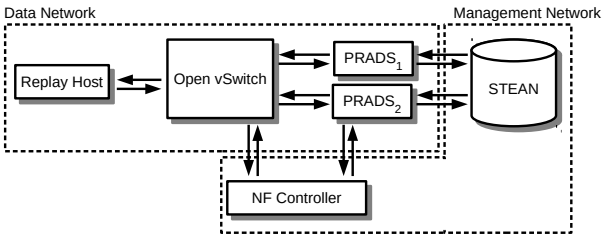

Fig. 6. Experimental setup for the evaluation of UC1.

We replay the trace at 500 packets per second and initially send all traffic to $PRADS_1$. Once it has created state for 250 and 400 flows, respectively, we initiate the migration of the flow state to $PRADS_2$.

The state is either migrated via the controller (OpenNF) or by sharing the current context using STEAN and only signalling the migration via the controller. All migrations are executed with order preserving enabled and STEAN executes filter transformations to select the context of the flow that is currently migrated. Fig. 7 shows the migration time for one TCP flow, comparing the OpenNF implementation to STEAN.

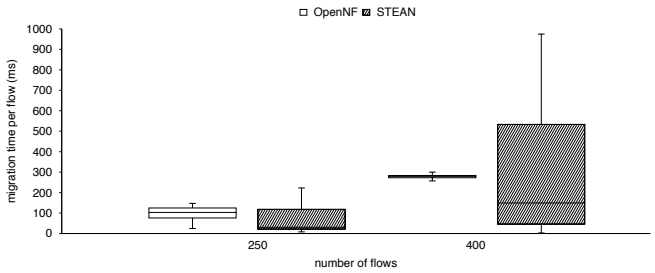

Fig. 7. Total migration time per flow context between two PRADS instances.

We observe that employing STEAN for context migration reduces the median migration time by 60 % per flow from 280 ms to 160 ms for the 400 flow case. This reduction, however, comes with an increased variability that is due to database locking of the current STEAN storage backend on inserts, delaying some concurrent requests.

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 $PRADS_2$, while the times for operations involving STEAN remain almost constant as shown in Fig. 8. The overall performance decrease

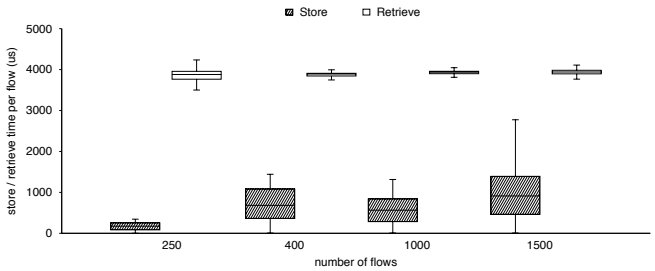

Fig. 8. Store and retrieve time per flow for migrating flows using STEAN.

when more than 400 flows are migrated is therefore not due to a bottleneck in STEAN but originates from either a congestion in the management network, or an overload of the controller.

Our results show that sharing context using STEAN is faster than migrating state employing OpenNF for the use case UC1, proving that STEAN can compete with state-of-the-art systems for migrating NF state.

C. UC3. Switching Routing Protocols: The main objective of this experiment is to demonstrate the applicability of transformation functions by providing seamless transitions between two routing protocols using STEAN.

We use a wireless mesh testbed in an office environment to conduct our experiments. Each host runs an instance of the modified AODV and OLSR implementations along with a STEAN instance to manage context. The STEAN instance

is configured with the base context and the transformation functions described in Section V-C2.

The experiments are conducted with a constant packet rate of 250 packets per second and a packet size of 1000 Bytes. The performance of STEAN does not depend on the absolute throughput of the network but rather on the packet rate as the number of requests to STEAN does not increase when using larger packets. We enabled client-side as well as STEAN-side caches for optimal forwarding performance.

First, we evaluate the behavior of OLSR when running the original implementation as well as our modifications that enable context sharing. Running OLSR with STEAN support to manage the protocol context increases the average end-to-end delay from 6.76 ms to 11.93 ms and the jitter from 1.76 ms to 103.08 ms. However, this increase is still acceptable for almost all applications running across a wireless mesh network and even allows for Voice over IP calls [12]. Here we observed that for $2/3$ of all packets the forwarding time is equal for both the standard and the STEAN-enabled implementation. The higher delay for other packets is due to blocking updates of the routing table that include packet counters and are thus altered regularly. Additionally, 95 % of all packets arrive within 33 ms and the high jitter comes from a few outliers.

Next, we execute a routing protocol transition from OLSR to AODV during runtime. The transition is triggered using a central controller and an out-of-band connection to each host as described in [13]. In Fig. 9, we show that the transition is executed without interruption in packet forwarding. The only visible effect is that the jitter is reduced and thus a better overall network performance is achieved.

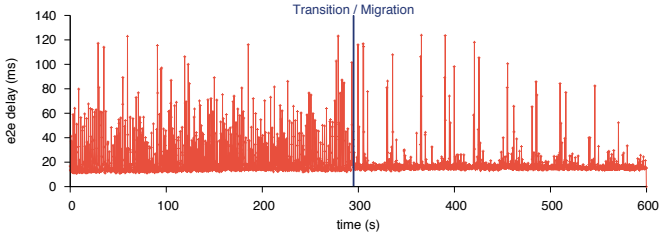

Fig. 9. End-to-end delay when migrating from OLSR to AODV during normal network operation. The transition is executed after 295 s (dark blue line).

We conclude from the experiments shown above that a common state store in conjunction with transformation functions as implemented by STEAN is able to support seamless protocol transition in WMNs with minimal overhead and enables protocol transitions during run-time without loosing end-to-end connectivity. The transition is transparent to end systems as well as overlying protocols.

D. Implementation Overhead: To quantify the modification required to support STEAN in the aforementioned systems, we counted the Lines of Code (LoC) that were added or changed in each implementation (Table I).

The results show that systems designed to share context information only require minimal changes to support STEAN. While the number of LoCs for AODV and OLSR might indicate rather dramatic changes, the actual implementation overhead was minimal since only a few functions needed to be changed. As these functions were largely scattered over the code, they increased the overall LoC count.

TABLE I
ADDITIONAL OR CHANGED CODE TO IMPLEMENT STEAN SUPPORT.

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

VII. Related Work

One approach of existing work to migrate NFs is moving the complete virtual machine (VM) as done by Remus [14]. This guarantees seamless failure recovery without modifying the function itself. However, migrating the VM comes at a significant cost as not only the relevant state of the NF itself but the state of the complete operating system is transferred to the backup system on a regular basis. Depending on the load of the actual VM, this *checkpointing* can cause a significant amount of additional latency to the normal operation. Alternatively, lightweight VMs such as specialized single-process containers can be used to reduce the overall replication overhead as shown by Tardigrade [15].

While migrating the complete (lightweight) VM suffices in a failover scenario, the systems lack an efficient measure to scale on different loads as the complete state of all flows needs to be duplicated, which not only wastes memory but also might result in a false behavior of the replicated NF.

Split/Merge [1] and Pico Replication [16] provide a framework to copy, migrate and replicate the state of NFs. They allow the migration of state from several instances of the same NF when creating or destroying a copy, or in the case of failover. In addition to the above, OpenNF [2] provides coordinated control of forwarding state in SDN to avoid packet loss or re-ordering, which can lead to a degraded performance of NFs. Kothandaraman et al. [17] as well as Gember-Jacobson and Akella [18] improve the performance of OpenNF by exchanging the function state directly without involving the control plane during migration. In contrast, Kablan et al. [3] propose to keep the state externally, leaving the NF itself stateless and centralizing all state management.

Statesman [19] introduces a network-wide state management architecture that is tailored towards data centers. It focuses on the collection and migration of states 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 resolving during state migration. Statesman focuses on the network-wide configuration state of management applications, but is not designed to handle the state of protocols or NFs.

TABLE II
OVERVIEW OF THE RELATED WORK.

	Scope		State Exchange		Features			Use Case		
	VM	App.	Direct Sharing	Migration	Persistence	Transformations	Decoupled State	NFs	Mgmt.	WMN
Remus [14]	++	−−	o	−	−−	−−	−−	+	o	−−
Tardigate [15]	++	−−	−−	+	−−	−−	−−	+	o	−−
Split/Merge [1]	−−	++	−−	+	−−	−−	−−	++	−	−−
Pico Replication [16]	−−	++	+	o	−−	−−	−−	++	−	−−
OpenNF [2]	−−	++	−−	+	o	o	−−	++	−	−−
DiST [17]	−−	++	−	++	−−	−−	−−	++	o	−−
p2p OpenNF [18]	−−	++	−	++	−−	−−	−−	++	o	−−
Stateless NF [3]	o	++	++	−	o	−−	++	++	o	−−
Statesman [19]	−−	++	+	−−	++	−	++	−	++	−−
STEAN	+	++	++	o	o	++	+	+	o	++

Table II gives an overview of the related work discussed above. It specifically shows that existing solutions focus on the migration of either the complete state of a VM (Remus, Tardigate) or the state of the NF running within this machine (Split/Merge, Pico Replication, OpenNF, Statesman).

STEAN separates the context from the actual functionality and provides a backend store for state information. While Stateless NF follows the same approach, our solution is also capable of storing state information from different protocols and applications across the network stack in a common base context and is thus capable of sharing the complete virtual machine state if required.

The current solutions provide state migration between systems of the exact same type as they directly extract the state from within the NF (Split/Merge, OpenNF, Stateless NF) or even require connecting applications to adopt to the state model of the management system (Statesman). To overcome this limitation, STEAN uses transformations to allow clients to specify their context and share information across implementations without adapting to a specific state model.

Furthermore, the existing systems focus on a single use case while STEAN specifically targets the complete network environment and provides a generalized solution for managing context information.

VIII. CONCLUSION

Sharing context information across components is essential for a more flexible and dynamic network management, and further boosts the deployment of new and innovative services. With this, we are able to overcome the limitations of current network functions and to include legacy systems such as routing protocols into new network architectures. Transformations are a core enabler for this extensive sharing as they allow us to support a large variety of network components and protocols without the need to adapt the internal state of these systems but with minimal changes to existing implementations.

We presented STEAN, a Storage and Transformation Engine for Advanced Network Context, that enables us to not only share state between instances of the same implementation but to extend the sharing of networking context beyond these boundaries. STEAN supports transformation functions by design and the architecture is centered around this core feature. Our evaluation shows that we are able to support a wide range of use cases with an acceptable overhead.

ACKNOWLEDGMENT

We thank our shepherd Olivier Bonaventure and the anonymous reviewers for their insightful comments and suggestions. This work has been co-funded by the DFG as part of the CRC 1053 MAKI and by LOEWE CASED.

REFERENCES

[1] S. Rajagopalan *et al.*, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes," in *NSDI*, 2013.
[2] A. Gember-Jacobson *et al.*, "OpenNF: Enabling Innovation in Network Function Control," in *SIGCOMM*, 2014.
[3] M. Kablan *et al.*, "Stateless Network Functions," in *HotMiddlebox*, 2015.
[4] E. Fjellskål, "Passive Real-time Asset Detection System," http://gamelinux.github.io/prads/.
[5] A. Osseiran *et al.*, "The Foundation of the Mobile and Wireless Communications System for 2020 and Beyond: Challenges, Enablers and Technology Solutions," in *VTC Spring*, 2013.
[6] M. Afanasyev *et al.*, "Analysis of a Mixed-Use Urban WiFi Network: When Metropolitan becomes Neapolitan," in *IMC*, 2008.
[7] M. Kalicinski, "RapidXML," http://rapidxml.sourceforge.net/.
[8] C. Perkins, E. Belding-Royer, and S. Das, "RFC 3561: Ad hoc On-Demand Distance Vector (AODV) Routing," IETF, RFC, 2003.
[9] P. Jacquet and T. Clausen, "RFC 3626: Optimized Link State Routing Protocol (OLSR)," IETF, RFC, 2003.
[10] E. Kohler *et al.*, "The Click Modular Router," *ACM TOCS*, vol. 18, no. 3, 2000.
[11] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *HotNets*, 2010.
[12] M. Karam and F. Tobagi, "Analysis of the Delay and Jitter of Voice Traffic Over the Internet," in *INFOCOM*, 2001.
[13] M. Werner *et al.*, "A Blueprint for Switching Between Secure Routing Protocols in Wireless Multihop Networks," in *WoWMoM*, 2013.
[14] B. Cully *et al.*, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *NSDI*, 2008.
[15] J. R. Lorch *et al.*, "Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services," in *NSDI*, 2015.
[16] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico Replication: A High Availability Framework for Middleboxes," in *SOCC*, 2013.
[17] B. Kothandaraman, M. Du, and P. Sköldström, "Centrally Controlled Distributed VNF State Management," in *HotMiddlebox*, 2015.
[18] A. Gember-Jacobson and A. Akella, "Improving the Safety, Scalability, and Efficiency of Network Function State Transfers," in *HotMiddlebox*, 2015.
[19] P. Sun *et al.*, "A Network-state Management Service," in *SIGCOMM*, 2014.