RESCue: A State-Disaggregated NFV System with Resilience, Elasticity, and State Consistency

Hao Jiang*, Hyunseok Chang†, Sarit Mukherjee† and Jacobus Van der Merwe*
* School of Computing, University of Utah, Salt Lake City, United States
† Network Systems and Security Research Department, Nokia Bell Labs, Murray Hill, United States

***Abstract*—State-disaggregated Network Function Virtualization (NFV) architectures decouple NF states from packet processing logic to achieve elasticity and resilience in stateful NFs. However, the existing state disaggregation approaches suffer from either poor NF performance due to frequent remote state access or potential inconsistencies in state updates when multiple NF instances concurrently access shared states. Moreover, they do not properly support state rejuvenation/expiration which is required for resource scalability of stateful NF operations. This paper presents a new state-disaggregated NFV system called RESCue that addresses these problems. RESCue handles remote state access differently for shared and private states. For efficient and consistent access of shared states, it leverages a lightweight custom control message protocol between NFs and a centralized state server. For private state access, it adopts a remote-paging-based interface to avoid introducing expensive blocking remote access within the critical path of NF packet processing. Finally, it utilizes non-blocking operations for state rejuvenation/expiration handling to minimize its performance overhead. Our evaluation of a RESCue prototype shows that it can handle NF scaling and failure recovery well, while supporting consistent state updates and state rejuvenation/expiration without compromising performance.**

I. Introduction

As a key component of modern data center infrastructure, NFs perform a variety of processing on data center traffic, such as packet header modification (load balancer, NAT), filtering (firewall), security inspection (intrusion detection system), encryption/decryption (SSL proxy), etc. NFV technology enables data center operators to realize these NFs as virtualized software instances running on general-purpose commodity servers, thereby significantly improving flexibility and elasticity in their deployments compared to dedicated hardware middleboxes [1]–[3].

One major challenge in the successful adoption of NFV in existing NFs is the need to support dynamic states within NFs [4], [5]. Unlike static states (e.g., NAT translation rules, firewall's access control policies), which are created a priori and do not change as traffic passes through NFs, dynamic states (e.g., NAT mappings, firewall's connection tracking states, load balancer's backend mappings, and usage statistics) are created based on arriving traffic, are constantly updated by NFs, either on a per-packet or per-flow basis, and may expire based on the freshness of these states.

A successful NFV system is obliged to not only (i) handle dynamic state updates in a highly performant manner for elasticity and resilience, but also (ii) preserve the correctness of the states when multiple NF instances concurrently update the same shared states during scale-out periods, and (iii) handle timely rejuvenation/expiration of the states as required by NFs. Achieving all three requirements adds significant complexity to state-disaggregated NFV realization but is critical for performant and correct NF operation. Specifically, if multiple NF instances read/write the same shared states concurrently without proper synchronization, state inconsistency can occur, possibly leading to incorrect NF operations. We will describe this problem in more detail in Section III. State rejuvenation and expiration are common requirements in modern NF implementations. After all, NF state capacity is not unlimited. States not only take up NF memory but also, in some cases, can be treated as scarce data center resources, e.g., a pool of available public IP addresses or port numbers in NAT. Therefore, states of old, inactive flows need to be constantly removed to make room for new flows. When incoming flows are a mixture of long-lived and short-lived flows, their corresponding states need to be refreshed periodically (i.e., rejuvenated) so that only those of short-lived flows age out.

There have been numerous research proposals that advance the state-of-the-art NFV systems. Some early efforts [1], [2], [12]–[14] achieve good performance, but do not support shared states. Others [6]–[8], [15], [16] design NFV systems allowing states to be migrated to newly created NF instances. However, they must buffer traffic and incur extended service downtime during state migration. Moreover, they are not resilient to failures.[1] StatelessNF [9] is the first attempt to disaggregate an NF's states from its packet processing logic. Although this decoupled design easily achieves elasticity and resilience, it suffers from increased per-packet latency and degraded packet processing throughput in NFs because every state operation requires blocking access to a remote memory store. Inspired by StatelessNF, S6 [10] and DAL [11] utilize techniques adopted from distributed systems to improve NF performance while maintaining elasticity. However, S6 still introduces stop-the-world downtime, and DAL has a long transient period during state migration in scale-out events. Additionally, neither S6 nor DAL is resilient to failures when there is no scale-out event, or when all NF instances of a given auto scaling group reside on one server. All these state-disaggregated efforts fail to address the state inconsistency problem and lack support for efficient state rejuvenation and expiration.

[1] Throughout this paper, when referring to failures, we mean both node and NF failures.

Proposal	Elasticity	Resilience	Performance	State consistency
E2 [2], SIMPLE [1]	No	No	Good	Yes
Split/Merge [6], OpenNF [7], ScaleFlux [8]	Yes	No	Extended downtime when migrating states	Yes
StatelessNF [9]	Yes	Yes	Significantly degraded latency/throughput performance	No
S6 [10]	Yes	No	Downtime when migrating states	No
DAL [11]	Yes	No	Transient period when migrating states	No

TABLE I: Summary of previous NFV proposals.

In this paper, we propose a new state-disaggregated NFV system called RESCue that not only provides Resilience and Elasticity for NF deployments, but also guarantees State Consistency and supports state rejuvenation and expiration without sacrificing performance. To minimize the performance implication of state disaggregation in RESCue, we rely on two guiding design principles: (i) minimize the number of blocking remote state accesses during which NF's packet processing has to be paused, and (ii) make each inevitable blocking remote access as efficient as possible. Following these principles, we categorize NF states into *shared states* (accessed by all NF instances of each auto-scaling group) and *private states* (accessed by only one NF instance). A typical NF instance has both shared and private states.

With state disaggregation, concurrent read/write access to shared states hosted by a remote state server is a major hurdle to achieving good performance while ensuring correctness. Our key design approach to handling shared states is to introduce *indexing* on shared states. We devise a state manager on the remote state server, which manages indexes for individual shared states and performs state operations such as state assignment, updates, rejuvenation, and expiration on behalf of NFs via these indexes. In this design, shared state access does not involve transferring actual states between NFs and the remote state server, but only introduces control messages that carry the indexes. The state manager maintains a dedicated thread for each NF instance to receive control messages from the instance. The control messages encode which state to assign or which state to update and how, and the state manager performs corresponding actions on behalf of NFs in their respective threads in a centralized fashion. This approach allows the complex distributed synchronization of shared states to be converted into thread-level local synchronization within the state manager. The overhead of local synchronization is negligible compared to that of its distributed counterpart. The frequency of control message communication can be adjusted based on the consistency requirements of given state operations. For example, for state rejuvenation and traffic statistics-related states which do not require strict per-packet updates, NFs can rate-limit the transmission of control messages. In summary, RESCue reduces the latency overhead for shared state access via lightweight control-message-based communication while ensuring consistency via synchronization within the state manager.

Unlike shared states, private states do not involve blocking remote interactions. RESCue adopts a remote-paging-based interface for accessing private states, so that the states can be accessed from local memory pages and automatically backed up to the remote state server outside the critical path of NF packet processing.

We utilize Remote Direct Memory Access (RDMA) to realize state access interfaces for shared and private states. For shared state access, we leverage the *RDMA WRITE with immediate* in an unconventional manner: transmitting an empty message with the four-byte immediate field to convey control messages. Remote paging for private state access is achieved via RDMA-backed network block device. We abstract low-level RDMA operations into high-level APIs to simplify NF developments for RESCue.

Our contributions are threefold. First, we present the design of RESCue to meet the requirements of state-disaggregated NFs in terms of correctness (state consistency) and features (state rejuvenation/expiration). Second, we implement a proof-of-concept prototype of RESCue for three popular NFs: NAT, firewall, and load balancer. Third, we extensively evaluate the prototype and show that RESCue achieves elasticity and resilience for NFs with dynamic states while supporting state rejuvenation/expiration and state consistency without compromising performance.

II. Related Work

For an NFV system to be deployable for real-world NF implementations, it must fulfill the following requirements: elasticity, resilience, high performance, and state consistency. We already discussed prior NFV frameworks [1], [2], [8]–[11] in terms of these requirements in the introduction and summarize their differences in Table I. *RESCue aims to satisfy all these requirements.*

Several earlier efforts focused on NF state management for state-disaggregated NFs. FlexState [17] proposes a highly configurable abstraction layer for NF states based on extended key-value interfaces. Its goal is to reduce code refactoring efforts in NF state management, thereby facilitating the rapid development of NFs. RedKV [18] proposes an RDMA-accelerated key-value store that facilitates elastic scaling for stateful NFs. Instead of relying on restrictive key-value interfaces, RESCue utilizes a more general memory-page-based state store. RECANS [19] adopts hierarchical state sharing for NFs, where NF state access can occur via either shared-memory-based local access or RDMA-based remote access, depending on the locality of data stores. Accordingly, access to cross-flow shared states is coordinated based on local vs. remote locking. On the other hand, RESCue avoids expensive

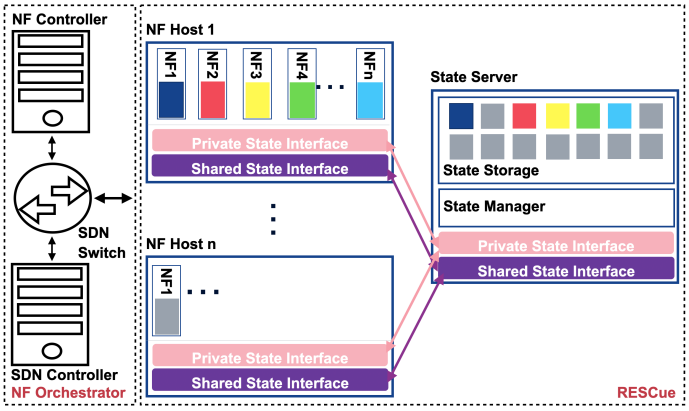

Fig. 1: RESCue architecture.

locking mechanisms for shared remote states and leverages a custom communication protocol with reduced round trip interactions. The proposed hierarchical sharing approach is not exclusive to RECANS, but can be adopted by RESCue as well. The authors of [20] consider algorithmic solutions for deciding where to place NF states and how many copies of states to maintain for short state access time and low network utilization. This work is orthogonal to RESCue.

While not directly proposed for NFV systems, some recent efforts [21], [22] present high-performance lock services for distributed systems by exploiting RDMA-based fast network transport. Instead of relying on such generalized and expensive locking mechanisms, RESCue leverages a control-message-based protocol specifically optimized for NF states supported by RESCue to minimize remote interactions.

III. RESCue Architecture Design

In this section, we present the design of RESCue. Its overall architecture is depicted in Figure 1. We assume that the NF orchestrator, which consists of the NF controller and the SDN controller, is already in place and thus not part of RESCue design. The NF controller deploys NF instances on end-hosts and handles NF failover/auto-scaling (i.e., spawning a new NF instance upon a failure in an existing instance and adding/removing NF instances based on their workloads). The SDN controller, in coordination with the NF controller, performs flow-level load balancing among NF instances in each auto-scaling group. Under this orchestration infrastructure, RESCue provides NF instances with custom state access interfaces, through which the instances dynamically create and place their shared/private states on state storage within the centralized state server. The state manager on the state server is responsible for conducting bookkeeping operations on hosted states on behalf of NFs. In the rest of the section, we describe these components in detail.

A. State Management

RESCue categorizes dynamic NF states into shared and private states based on their access patterns. Examples of shared states include a pool of shared resources distributed to a group of NF instances (e.g., a pool of available public IP addresses/ports in NAT) or global traffic statistics cumulatively updated by all NF instances in each auto-scaling group. On the other hand, private states are those accessed by only one specific NF instance, typically related to flows assigned to the instance (e.g., flow-level NAT mappings or firewall's connection tracking states). In the following, we describe how RESCue handles these states differently.

1) Shared States: Shared states can be frequently accessed and updated simultaneously by all NF instances in a given auto-scaling group. In RESCue, we manage them at a centralized state server, as opposed to distributing and dynamically migrating them across NF instances (e.g., [10], [11]). The centralized placement of shared states has the following advantages. First, since states do not need to be migrated across NF instances, there is no migration overhead nor risk of frequently bouncing shared states back and forth among instances. More importantly, there is no need to sacrifice state consistency for performance since the centralized state server can handle proper synchronization, as will be described later.

As a downside, this approach requires that NF instances perform remote interactions with a centralized state server during shared state access and during state rejuvenation/expiration events. To mitigate this issue and enhance performance, we minimize the frequency of these remote state accesses by optimizing state access operations based on their requirements. Table II summarizes the types of shared states maintained by popular NFs. These states are largely broken down to either shared resources or global statistics. Ensuring strict consistency in updating shared resources among NF instances is particularly important for correct NF operations. On the other hand, updating global statistics does not require as strict consistency since the statistics typically are not tied to the correctness of NF operations. Recognizing these requirements, we devise lightweight control-message-based communication between NFs and the centralized state server to manage shared states in an efficient and consistent fashion. Section III-B provides more detail on this topic.

NF type	Shared states	Category
NAT	Public IP address (port) pool	Resources
Firewall	Packet or flow counters	Statistics
Load balancer	Per-backend usage statistics	Statistics
IDS/IPS	Lists of malicious nodes, infected servers, etc.	Statistics
Traffic monitor	Statistics for packets, protocols, hosts, etc.	Statistics

TABLE II: Shared states of popular NFs.

2) Private States: As described earlier, RESCue is designed under the assumption that the SDN controller performs flow-level load balancing among NF instances in each auto-scaling group. Thus, flow-level states are not shared by multiple NF instances, but remain private to each respective instance. To support accessing such private states efficiently, we adopt a remote-paging-based access interface. Remote paging allows remotely hosted private states to be loaded to local memory pages (after page faults) and subsequently accessed by NF instances locally. Any state updates on dirty pages are auto-

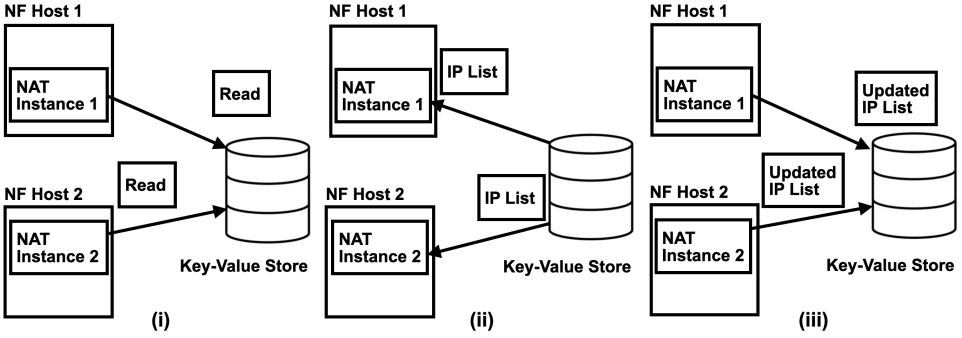

Fig. 2: The state inconsistency problem with NAT.

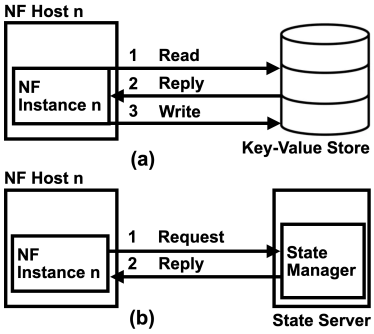

Fig. 3: State store comparison.

matically backed up to the remote state server on memory page granularity outside the critical path of NF packet processing. To avoid frequent small updates, RESCue allows batched updates. Both the memory page size and batching interval can be fine-tuned based on the size of per-flow NF states and state access patterns. Compared to key-value-based interfaces adopted by previous state-disaggregated efforts [9]–[11], this interface, thanks to its memory access semantics, allows NFs to export and access more complex types of NF states in a more generalized fashion and with fewer round trips.

B. Consistency in Shared NF States

Maintaining consistency in shared states is highly challenging in NFV systems, especially with state-disaggregated approaches. Early efforts [6], [7] sacrifice performance to maintain state consistency, while other state-disaggregated NFV proposals trade consistency for performance [10], or do not address the problem at all [9], [11]. In Figure 2, we illustrate the state inconsistency problem with NAT as an example. In this scenario, two NAT instances concurrently access a pool of available public IP addresses/ports as a shared state stored on a remote key-value store. Upon receiving a packet for a new flow f_1, instance 1 issues a read request to the key-value store to obtain the list of currently available public IP addresses/ports. At the same time, instance 2 issues the same read request for a new flow f_2. The key-value store replies to both instances with the same public IP addresses/ports list. Each instance selects an available public IP address/port from the list and issues a write to update the list on the remote store, one overwriting the other. Depending on the selection algorithm used or the pool size, they may or may not pick the same IP address/port. Even when they choose different ones, instance 2 will still overwrite the update on the list made by instance 1, allowing the IP address/port picked by instance 1 to become available for other instances. Either way, multiple outgoing flows may be mapped to the same public IP address/port by NAT, leading to incorrect NAT operation. In typical scale-out NFV deployments, correctness is as essential as performance, so this type of state inconsistency is not acceptable. In the following, we describe how RESCue achieves state consistency with minimal performance overhead.

To avoid data races that cause state inconsistency, synchronization (e.g., using locking mechanisms) is necessary. However, distributed synchronization in distributed systems incurs a tremendous negative impact on performance. Our goal in designing shared state management in RESCue is to offload distributed synchronization overhead from NF instances and pass it onto the centralized state server while minimizing state transmission overhead. This motivates us to adopt custom control-message-based communication between NF instances and the state server for shared state management. In this design, we introduce numeric indexes for individual shared states. The custom control messages that carry the indexes and an "instruction" (OpCode) are exchanged between NFs and the state server in order to tell the state server which state to assign or which state to update and how, thereby allowing it to perform necessary state operations on behalf of NFs without transmitting actual states between them. As already shown in Table II, there are largely two types of shared states. For shared resources, the state server handles their assignment, rejuvenation, and expiration, while, for statistics, the state server manages their update and lookup.

Control message handling on the state server is performed by the state manager which is responsible for processing consistent updates on shared states among deployed NF instances. The state manager spawns a dedicated thread for each NF instance, which interacts with the instance via control messages to perform shared state processing on its behalf. When multiple NF instances access the same shared state, their respective threads within the state manager perform corresponding state operations locally in a thread-safe manner. This approach turns otherwise performance-demanding distributed synchronization of shared states into lightweight thread-level local synchronization.

The state manager performs two types of shared state operations: passive and active. Passive operations are performed upon receiving control messages from NF instances, such as assigning and rejuvenating indexes for shared resources and updating and reading traffic statistics. Active operations are used to actively release expired resources and make them available again for new assignments. More detail on state rejuvenation and expiration is found in Section III-C.

C. State Rejuvenation and Expiration

Real-world NF implementations require old states to expire after a certain period of inactivity and active states to

be retained indefinitely via a state rejuvenation mechanism. Without proper support for state rejuvenation and expiration, it may lead to resource exhaustion in NFs (e.g., depleted address pool in NAT) or sometimes incorrect NF operations (e.g., long-lived flows accidentally dropped by NAT/firewall). StatelessNF [9] uses TCP/FIN to signify the expiration of a flow and its states. However, such a protocol-dependent approach has the following drawbacks. If an attacker sends TCP/SYN packets for a large number of flows, but without corresponding TCP/FIN packets, the capacity of NF instances will easily be exhausted, leading to a denial of services on normal traffic. Without state expiration, the remote state stores will be filled with stale states that will never be cleared. This approach also cannot handle the expiration of UDP flows without TCP/FIN-equivalent signaling packets. Other state-disaggregated approaches like S6 [10] and DAL [11] do not handle state expiration at all.

In RESCue, we adopt the following state rejuvenation/expiration approach to solve the problem. First of all, since private states are accessed transparently from local memory pages, state rejuvenation and expiration for private states can easily be handled by native NF implementation. Thus, we focus on shared states managed via custom control messages described earlier. In a nutshell, state expiration is handled by the state manager, while state rejuvenation is initiated by NF instances. In both cases, the state manager and NF instances utilize control messages to communicate the outcome of state rejuvenation/expiration to the other party. Note that, among shared states, state rejuvenation and expiration are only applicable to shared resources, not statistics-related states. For state expiration, the state manager maintains a table recording which index values (and corresponding share resources) are assigned to which NF instances. When the assignment of a given shared resource expires after a configurable timeout, the state manager clears the index from the table and communicates this event to an associated NF instance. On the other hand, state rejuvenation is initiated by NF instances. When a state needs to be rejuvenated, an NF instance sends a control message carrying the corresponding index to the state manager, instructing it to rejuvenate this state on the state server.

Since state rejuvenation and expiration for shared states increase the frequency of remote interactions between NF instances and the state store, it can lead to degraded NF performance. RESCue provides several optimizations to mitigate the impact of these operations. First, the shared state interface for state rejuvenation and expiration (elaborated in Section IV-A) is realized as non-blocking operations. Second, RESCue allows the frequency of performing rejuvenation operations to be configurable. State rejuvenation for a given flow can be triggered by a mix of a packet count threshold and the last rejuvenation timestamp. Since the goal of state rejuvenation is to properly support long-lived flows (regardless of state expiration threshold), reducing its frequency does not impact the correctness of NF operations.

D. State Store

Our state server design for shared and private states is much more than simple key-value stores which have limitations in supporting different types of states [9]. Not all types of states are suitable to be accessed via a key-value interface as adopted by [9]–[11]. The state interfaces of RESCue for shared and private states allow NF developers to store dynamic states in a more consistent and generalized fashion.

In addition, the state manager on the state server simplifies the way shared states are assigned to make it more efficient. Figure 3 compares the ways per-flow states are accessed by the architectures adopting key-value stores and our approach. In the case of key-value stores, an NF instance has to issue a remote read to check which resources are available, followed by a remote write to obtain one. The process involves at least one and a half round trip interactions with the key-value store. In our design, on the other hand, an NF instance issues a request for the next available resource, and the state manager responds with either an assigned resource or a notification that none is currently available. This process is completed in only one round trip communication. In terms of state server overhead, key-value stores have lookup overhead for both read and write operations since each state access involves hashing of the key. Whereas in RESCue, the state server simply loads memory pages into NF instances, and private state access operations are performed by NF instances locally as regular memory accesses, saving computation power and time on the state server.

IV. Implementation

As a proof of concept, we implement a RESCue prototype. In the following, we highlight its key implementation details.

A. State Access Interfaces

The performance of RESCue is predicated on the efficiency of state access interfaces described in Section III. To implement the interfaces with efficiency and low latency in mind, we adopt RDMA as the underlying network transport. For the shared state access interface, we devise a novel RDMA-based messaging mechanism by applying unconventional use of RDMA verbs. To enable RDMA-based remote paging for the private state access interface, we explore several open-source solutions, and in the end adopt the RDMA-backed network block device (RNBD) [23] as it is the most efficient and stable with community support from mainline Linux kernel integration (starting from v5.8). The following subsections describe the details.

1) Shared State Access Interface: In implementing control-message-driven shared state access with RDMA, there can be multiple options for underlying RDMA operations. Table IV lists these options and their features. While one-sided WRITE achieves the best performance with receiver-side CPU bypass [24], it is not an ideal choice for the following reason. Without explicit notification for incoming WRITE, the receiver has to allocate memory regions for incoming control messages and constantly poll the regions to consume the messages.

OpCode (7 bits)	Index list (5 bits)	Indexes (20 bits)	Action description
INDEX_REQUEST (1)	3	0	Assign the next available index in list 3.
INDEX_ASSIGNMENT (2)	3	7000	Index value of 7000 in list 3 is assigned in response to NF's INDEX_REQUEST.
NO_MORE_INDEX (3)	3	0	If there is no available index to assign in list 3, reply this OpCode to an NF.
UPDATE_STATISTICS (4)	5	36000	Add 1 to the statistics state indexed 36000 in list 5.
UPDATE_FAILURE (5)	5	36000	Fail to update the statistics state indexed 36000 in list 5. Reply this OpCode to an NF.
EXPIRE (6)	2	5000	The state indexed 5000 in list 2 has expired. Send this OpCode to the NF that was previously assigned it.
REJUVENATE (7)	9	100001	Rejuvenate the state indexed 100001 in list 9.

TABLE III: Examples of control message encoding for OpCode and indexes.

RDMA operation	Features
SEND	Receiver posts RECEIVE before Sender transmits SEND. Receiver is notified.
WRITE	No RECEIVE is posted. Receiver is not notified.
SEND with immediate	Receiver posts RECEIVE before Sender transmits SEND. Receiver is notified with a four-byte immediate field. RECEIVE is read.
WRITE with immediate	Receiver posts RECEIVE before Sender transmits WRITE. Receiver is notified with a four-byte immediate field. RECEIVE is not read.

TABLE IV: RDMA operations and their features.

When numerous control messages are received back to back, both RDMA NICs and receiver-side CPUs will attempt to update the same memory regions concurrently, and this can lead to complicated race conditions. In that sense, WRITE with immediate can be a better alternative to WRITE since the former sends a message together with out-of-band immediate data, which can trigger the receiver-side notification. The immediate data field is normally used as metadata of a transmitted message. In our implementation of the shared state access interface, we leverage WRITE with immediate in a non-traditional manner, where we transmit a *zero-byte* message with immediate data. This approach is driven by our design, where shared state communication does not transmit states but only carries control messages that are small enough to fit into four-byte immediate data. Compared with conventional non-empty WRITE with immediate, this approach bypasses two steps in the critical path of message transmission: (i) the sender-side RDMA NIC reading a message from the sender's memory and (ii) the receiver-side RDMA NIC writing a message into the receiver's memory. Compared to SEND with immediate, WRITE with immediate has a performance advantage because, in the former case, the receiver not only consumes an outstanding posted RECEIVE request but also reads it. To the best of our knowledge, RESCue is the first research work in the literature that applies the novel approach of *zero-byte* RDMA WRITE with immediate.

Four-byte control messages exchanged between NFs and the state manager carry not just a numeric index for a shared state but also OpCode to instruct the state manager or NFs to perform a specific action on the shared state. Table III shows examples of control messages supported by RESCue.

2) Private State Access Interface: For remote-paging-based private state access, we use RNBD as the underlying remote block device. Upon start of an NF instance, it performs `mmap` on an RNBD device to store and access private states of its assigned flows within its address space. If the instance is started due to NF failover, previously stored flow states can be automatically loaded after page faults. On each NF server, we create separate RNBD devices for individual NF types (e.g., NAT, firewall, load balancer). When more than one NF instances of the same NF type are deployed on a server for NF scaling, the instances share the same RNBD device. Different instances will then access different memory-mapped regions of the device to store private states of their assigned flows. That way, flows assigned to one NF instance can be flexibly moved to another instance on the same server without affecting their execution. Such flow reassignment may be needed to minimize the impact of NF failover or due to dynamic flow rate changes. To allow NFs to access memory-mapped private states without significant code modification, we implement a custom memory allocation API called `rescue_malloc` which is to be used when developing NFs for RESCue. The standard `malloc` API is not suitable since it can only assign memory on the heap, and we cannot control the exact location of the assigned memory. The `rescue_malloc` API allocates memory for NF private states from the memory region mapped by RNBD, and at specified locations (typically consecutively for memory savings and efficient access).

B. Network Functions and State Server

We integrate the aforementioned state access interfaces into three existing stateful NFs available from VigorNF [25]: NAT, firewall, and load balancer. These NFs are userspace implementations developed with DPDK [26]. Our extension of the memory management library and APIs of VigorNF for RESCue is written in ∼500 SLOC of C. The modification of these NFs for RESCue integration comprises ∼900 SLOC of C (300 SLOC per NF). Through the library and APIs, we only need to add minor modifications to the original NFs. We also implement the state manager to communicate with each of these NFs via the shared state interface to enable state assignment, rejuvenation, and expiration. The state manager is written in ∼1500 SLOC of C. The state store

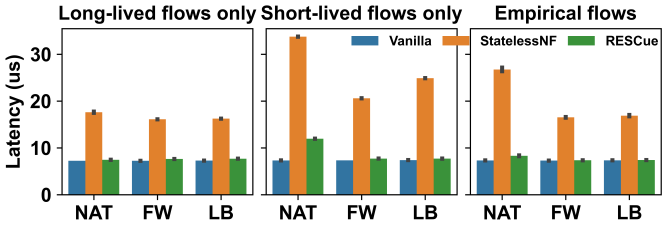

Fig. 4: Latency comparison with different flow types.

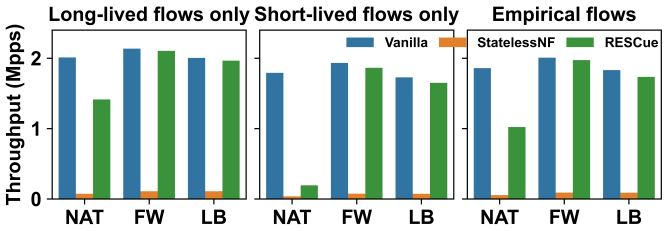

Fig. 5: Throughput comparison with different flow types.

where private states are stored via the RNBD server is set up using RAMDisk. To evaluate the prototype, we also develop a custom DPDK-based packet generator in ∼1300 SLOC of C, which can generate various types of flows (e.g., based on empirical flow distributions).

V. Evaluation

In this section, we present our evaluation of the RESCue prototype. In our evaluation, we aim to answer the following questions with regards to RESCue: (i) How does its performance compare with previous efforts? (ii) How does the support for state rejuvenation/expiration affect NF performance? (iii) Does it guarantee state consistency? (iv) Does it support NF scaling well? (v) Is it resilient to failures?

A. Experimental Settings

We deploy the RESCue prototype and run our experiments on a testbed consisting of three AMD EPYC Rome servers, each equipped with 16 CPU cores, 128 GB memory, and a 25 GbE dual-port Mellanox ConnectX-5 NIC, provisioned from the POWDER testbed infrastructure [27]. We designate one server to run a traffic generator, another to run NF instances, and the last one to run the state server. All three servers run on Ubuntu 20.04 with kernel v5.8. For NF scaling experiments, we enable SR-IOV on the NF server's NIC to run multiple NF instances on separate virtual function interfaces.

B. NF Latency/Throughput Performance

For state-disaggregated NFV systems like RESCue, NF performance can vary significantly depending on the type of traffic mix, especially for NFs with shared resources. Short-lived flows require frequent remote access for resource assignments and thus have a more significant impact on performance compared to long-lived flows. We measure NF performance in terms of latency and throughput under three different traffic conditions: (i) long-lived flows only, (ii) short-lived flows only, and (iii) an empirical mix of flows. We set the properties of these flows based on real-world data center traffic [28]. Each long-lived flow is composed of 1000 packets,

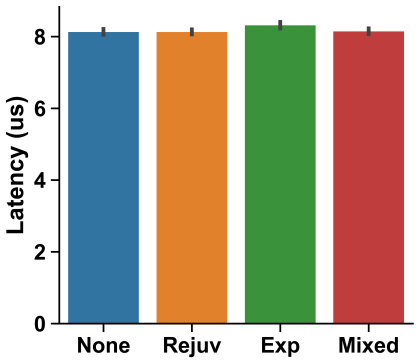

Fig. 6: Latency overhead of state rejuvenation/expiration.

while each short-lived flow carries 10 packets. Empirical flows comprise 20% long-lived and 80% short-lived flows. The scenarios (i) and (ii) serve as two extreme cases to benchmark NF performance. We use three NFs to evaluate performance: NAT, firewall, and load balancer. We compare RESCue's performance against two baselines. In "Vanilla", we use unmodified NFs from VigorNF [25], which perform stateful packet processing locally without state disaggregation. In "StatelessNF", we extend VigorNF to support state disaggregation as described in StatelessNF [9].

Figure 4 compares latency performance among three schemes. RESCue NFs exhibit comparable latency as vanilla NFs without state disaggregation and perform significantly better than StatelessNF. As expected, for both RESCue and StatelessNF, NAT experiences longer latency than its vanilla counterpart with short-flows due to more frequent shared resource assignments, but the latency penalty of RESCue is significantly smaller than StatelessNF due to its lightweight control-message-based communication. Figure 5 evaluates NF throughput performance. Firewall and load balancer on RESCue retain comparable performance as vanilla NFs, while RESCue NAT experiences a negative impact of remote resource assignments. The latter is because the VigorNF platform we adopt does not support multi-threaded packet processing. Thus the entire packet processing in NAT is blocked during per-flow resource assignment for consistency. Even with consistent assignments, RESCue significantly outperforms StatelessNF. The throughput difference is sizable because StatelessNF suffers more from the same single-thread limitation of the VigorNF platform since StatelessNF requires multiple remote state accesses for each new flow. StatelessNF [9] reports higher NF throughtputs than our implementation because their implementation uses multiple threads for parallel packet processing.

Unfortunately, we were unable to perform direct performance comparison between RESCue and previous efforts such as S6 [10] and DAL [11]. S6 cannot run on our platform due to incompatibility in their system requirements and application dependencies, and because they use different types of NFs than RESCue and StatelessNF. DAL is not open-source. Unlike RESCue, S6 and DAL distribute states to individual NF instances. Therefore, when there is no scale-out event or when all NF instances of a given auto-scaling group are co-located

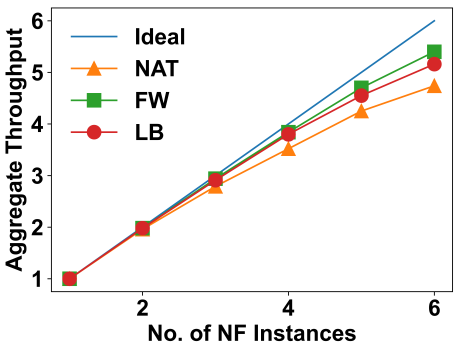

Fig. 7: Elasticity upon NF scaling.

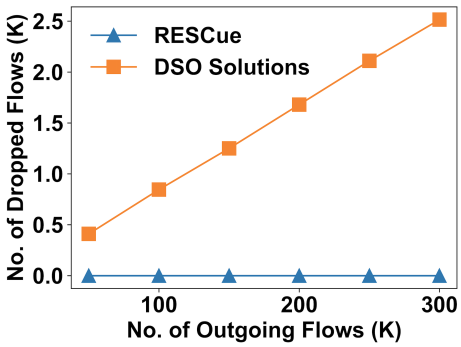

Fig. 8: State consistency.

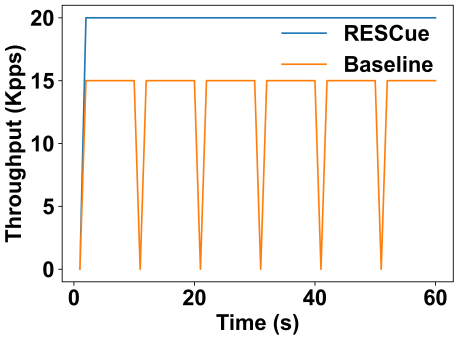

Fig. 9: Resilience to failures.

on one server, S6 and DAL will have better performance than RESCue since all states are served locally, while RESCue incurs the overhead of remote handling of shared states. However, S6 and DAL are not resilient to failures in this case since all states will be lost if the server where NF instances are hosted fails. When there are multiple NF instances deployed across different servers, RESCue is expected to have comparable performance as S6 and DAL, although they will still suffer from the overhead of migrating shared states among NF instances. Moreover, with multiple NF instances, S6 and DAL will be subject to potential state inconsistencies, and they do not support state rejuvenation/expiration. RESCue addresses all these issues.

C. Overhead of State Rejuvenation/Expiration

As described in Section III, RESCue supports state rejuvenation and expiration, and we realize this functionality with non-blocking RDMA operations. To evaluate the impact of state rejuvenation and expiration on performance, we measure NF latency for NAT on RESCue with empirical flows. We compare the following scenarios: (i) rejuvenation and expiration are disabled ("None"), (ii) per-packet state rejuvenation is enabled ("Rejuv"), (iii) per-packet state expiration is enabled ("Exp"), and (iv) rejuvenation and expiration are triggered for a batch of 10 packets ("Mixed"). Figure 6 shows that both rejuvenation and expiration operations incur negligible overhead on the latency of NAT due to their non-blocking nature. State expiration introduces slightly higher latency than rejuvenation. This is because the state expiration operation involves searching and removing private states in a hash table using a hashed key. This is more heavy-duty than the rejuvenation operation, which merely finds and updates a timestamp for an index.

D. NF Scaling

To explore how well RESCue supports NF scaling, we deploy NAT, firewall, and load balancer with a scaling group of up to six instances, each of which handles empirical flows. All NF instances are deployed on one server. Due to the limited number of NIC resources on the NF server, NF traffic and remote state access traffic share the same 25 GbE physical link. We measure NF performance in terms of latency and throughput as we increase the number of NF instances in the scaling group. We do not observe a noticeable impact on latency after scaling-out events. Thus we report only throughput results. Figure 7 shows how NF throughput scales with the number of instances for the three types of NFs. The linear line labeled "Ideal" is a reference where the aggregate throughput is computed as single-instance throughput multiplied by the number of instances. Relative to this hypothetical reference line, all three NFs experience throughput degradation after scaling-out events. We believe that multiple factors cause the throughput degradations. First, the overhead of supporting state rejuvenation/expiration and ensuring state consistency increases with the number of deployed NF instances. Second, sharing the same physical link for NF traffic and shared state access introduces additional contention on NIC resources.

E. NF State Consistency

Next, we deploy NAT function to demonstrate that RESCue can support consistent state updates. As illustrated earlier in Section III-B, without proper synchronization, multiple NAT instances concurrently accessing the same pool of public IP addresses and port numbers could result in a situation where multiple incoming flows are assigned the same public IP address/port. This will make returning (incoming) flows after NATing be directed to the wrong source hosts, leading to packet drops for these flows. In this experiment, we deploy up to six NAT instances and measure the number of dropped flows due to state inconsistency. Our packet generator generates traffic based on empirical flow distributions at a data rate close to the maximum throughput of NAT to each NAT instance. For comparison, we repeat the experiment after disabling the state consistency support, which emulates previous solutions based on distributed shared object (DSO) such as S6 and DAL. Figure 8 confirms that RESCue incurs no packet drop for returning flows due to the state consistency guarantee. On the other hand, DSO solutions suffer from non-negligible flow drop events, which increase proportionally with the number of flows.

F. Resilience to Failures

Finally, we demonstrate the ability of RESCue to recover from failures by using load balancer available from VigorNF. The load balancer NF has two types of private states: flow connection contexts and backend information. In VigorNF

implementation, backend instances send heartbeat messages at configurable intervals to the load balancer. Upon receiving a heartbeat message from a backend for the first time, the load balancer records information such as the IP/MAC addresses of this backend, as well as the NIC port number connecting to it. Then this backend is registered as an available one for flow load-balancing until it expires. Future heartbeat messages from the same backend rejuvenate its valid status. Upon receiving incoming flows, the load balancer directs traffic to available backends. If there are no available backends to assign, the packets are dropped. We use the unmodified load balancer without state disaggregation as the baseline for comparison. To measure the effect of disruption caused by failover, we assume perfect failure detection (i.e., restart a load balancer instance immediately upon failure). We run six rounds of experiments with RESCue-integrated load balancer and the baseline counterpart. After each round is completed, we terminate the currently running load balancer instance and instantiate a new one. Figure 9 shows the impact of failure events on data rates. For better visualization, we use different data rates for RESCue and the baseline. At time 1, both instances experience zero data rate because there are no available backends in them. At time 2, after heartbeat messages have been received, the load balancer in each case can process packets at steady data rates for incoming flows. At times 11, 21, 31, 41, and 51, the baseline experiences disruptions in data rate because a newly instantiated instance loses all backend states during failures. The data rate recovers only after new heartbeat messages have been sent at time 12, 22, 32, 42, and 52. In the case of RESCue, there is no further disruption after time 1 since all backend states are loaded from the state server through the private state access interface after each failure event.

VI. Conclusion

In this paper, we presented RESCue, a state-disaggregated NFV system that provides resilience and elasticity, enforces state consistency, and supports state rejuvenation/expiration without compromising performance. We described how our design addresses problems that previous efforts failed to do. Our extensive evaluation of the prototype demonstrated its advantages and capabilities. We believe that RESCue is a solid advance of state-disaggregated NFV to make it more feasible in modern data centers. In the future, we plan to apply RESCue to a more diverse and comprehensive set of real-world stateful NFs.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant Number 1827940.

References

[1] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM 2013*, 2013, pp. 27–38.

[2] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A framework for NFV applications," in *Proc. ACM SOSP 2015*, 2015, pp. 121–136.

[3] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[4] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. ACM Symposium on Cloud Computing*, 2013, pp. 1–15.

[5] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-recovery for middleboxes," in *Proc. ACM SIGCOMM '15*, 2015, pp. 227–240.

[6] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System support for elastic execution in virtual middleboxes," in *Proc. USENIX NSDI '13*, 2013, pp. 227–240.

[7] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2014.

[8] L. Liu, H. Xu, Z. Niu, J. Li, W. Zhang, P. Wang, J. Li, J. C. Xue, and C. Wang, "ScaleFlux: Efficient stateful scaling in NFV," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4801–4817, 2022.

[9] M. Kablan, "StatelessNF: A disaggregated architecture for network functions," Ph.D. dissertation, University of Colorado at Boulder, 2017.

[10] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. USENIX NSDI '18*, 2018, pp. 299–312.

[11] M. Szalay, M. Nagy, D. Géhberger, Z. Kiss, P. Mátray, F. Németh, G. Pongrácz, G. Rétvári, and L. Toka, "Industrial-scale stateless network functions," in *Proc. 2019 IEEE CLOUD*, 2019, pp. 383–390.

[12] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. USENIX NSDI '12*, 2012, pp. 323–336.

[13] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "FlowTags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proc. ACM HotSDN '13*, 2013, pp. 19–24.

[14] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," Tech. Rep., 2013.

[15] H. Jiang, N. Choi, M. Thottan, and J. Van der Merwe, "FestNet: A flexible and efficient sliced transport network," in *Proc. 2021 IEEE NetSoft*, 2021, pp. 97–105.

[16] E. Keller, J. Rexford, and J. E. van der Merwe, "Seamless BGP migration with router grafting." in *Proc. USENIX NSDI '10*, 2010, pp. 235–248.

[17] M. Pozza, A. Rao, D. Lugones, and S. Tarkoma, "FlexState: Flexible state management of network functions," *IEEE Access*, vol. 9, 2021.

[18] C. Chang, W. Yang, C. Zheng, P. Jiang, L. Zhan, and Q. Liu, "Accelerate state sharing of network function with rdma," in *Proc. IEEE GLOBECOM 2022*, 2022, pp. 1–6.

[19] J. Zhao, S. Zhuang, J. Li, and H. Guan, "RECANS: Low-latency network function chains with hierarchical state sharing," in *Proc. International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 209–220.

[20] M. Szalay, P. Mátray, and L. Toka, "State management for cloud-native applications," *Eletronics*, vol. 10, no. 4, 2021.

[21] D. Y. Yoon, M. Chowdhury, and B. Mozafari, "Distributed lock management with RDMA: Decentralization without starvation," in *Proc. 2018 International Conference on Management of Data*, 2018, pp. 1571–1586.

[22] Y. Chung and E. Zamanian, "Using RDMA for lock management," *arXiv preprint arXiv:1507.03274*, 2015.

[23] "Rdma network block device," https://github.com/ionos-enterprise/rnbd.

[24] P. MacArthur and R. D. Russell, "A performance study to guide RDMA programming decisions," in *Proc. 2012 IEEE HPCC*, 2012, pp. 121–136.

[25] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea, "Verifying software network functions with no verification expertise," in *Proc. ACM SOSP '19*, 2019, pp. 275–290.

[26] "Data plane development kit (dpdk)," https://www.dpdk.org.

[27] "Powder: Platform for open wireless data-driven experimental research," https://powderwireless.net.

[28] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM conference on Internet measurement*, 2010, pp. 267–280.