Received January 19, 2021, accepted February 18, 2021, date of publication February 24, 2021, date of current version March 30, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3061814

FlexState: Flexible State Management of Network Functions

MATTEO POZZA[1], ASHWIN RAO[1], DIEGO F. LUGONES[2], (Member, IEEE), AND SASU TARKOMA[1], (Senior Member, IEEE)
[1]Department of Computer Science, University of Helsinki, 00014 Helsinki, Finland
[2]Nokia Bell Labs, Dublin 15, D15 Y6NT Ireland

Corresponding author: Matteo Pozza (matteo.pozza@helsinki.fi)

This work was supported in part by the Nokia Center for Advanced Research (NCAR), and in part by the 5G FORCE Research Project.

ABSTRACT Network function (NF) developers have traditionally prioritized performance when creating new packet processing capabilities. This was usually driven by a market demand for highly available solutions with differentiating features running at line rate, even at the expense of flexibility and tightly-coupled monolithic designs. Today, however, the market advantage is achieved by providing more features in shorter development cycles and quickly deploying them in different operating environments. In fact, network operators are increasingly adopting continuous software delivery practices as well as new architectural styles (e.g., *microservices*) to decouple functionality and accelerate development. A key challenge in revisiting NF design is state management, which is usually highly optimized for a deployment by carefully selecting the underlying data store. Therefore, migrating to a data store that suits a different use case is time-consuming as it requires code refactoring and adaptation to new application programming interfaces, APIs. As a result, refactoring NF software for different environments can take up to months, reducing the pace at which new features and upgrades can be deployed in production networks. In this paper, we demonstrate experimentally that it is feasible to introduce an abstraction layer to decouple NF state management from the data store adopted while still approaching line-rate performance. We present *FlexState*, a state management system that exposes data store functionality as *configuration options*, which reduces code refactoring efforts. Experiments show that FlexState achieves significant flexibility in optimizing the state management, and accelerates deployment on new scenarios while preserving performance and scalability.

INDEX TERMS Data storage systems, middleboxes, network functions, network functions virtualization, NFV, parallel processing, state management.

I. INTRODUCTION

Network functions (NFs), such as network address translators (NATs), load balancers or intrusion detection systems (IDS) are stateful. For this reason, developers must deal with the inherent trade-off of maintaining a consistent state shared across packet flows manipulated by multiple NF instances while processing packets at line rate [1], [2].

This trade-off has become significantly more challenging with the adoption of cloud-native principles (CNF [3]) that enable efficient container packaging, continuous delivery and integration, decomposition, autoscaling, and off-the-shelf platform services like generic load-balancers and data stores. That is, transitioning to containerized deployments where NF instances can be created and terminated dynamically, and where NF components are interchangeable and loosely coupled, adds complexity to the state management since there is a need for new functionality to gracefully migrate network traffic, handle session-related variables and manipulate flow information end to end.

A first step to overcome this challenge is to delegate the management of NFs state information to dedicated systems. Such systems are designed and optimized for a specific set of requirements driven by conflicting needs, which determine the appropriate data store applicable to a specific use case. Some of these systems are, for example, *StatelessNF* [4], which relies on a remote key-value store (KVS) to provide reliability; or *S6* [5], which uses a distributed hash table (DHT) to optimize for high performance instead.

More concretely, NF operational requirements can vary quite significantly among use cases. For instance, a network tailored for stock trading targets the lowest achievable

The associate editor coordinating the review of this manuscript and approving it for publication was Zubair Md. Fadlullah.

latency [6], whereas for voice and video services, networks must be robust to disruptions [7]. Thus, the specific use case influences the selection of the data store used internally by the state management system, and in practice, developers need to design packet-processing functions that can be adapted to a variety of scenarios and work properly with different data store optimized features.

However, in NF development today most state management systems are tightly coupled to the specific data store used internally, and it requires a significant effort to incorporate new data stores or even upgrades of the existing one. That is, the selection of a data store and its optimization condition the state management API that is ultimately exposed to the packet processing logic of the network function. In turn, this also creates a dependency between the NFs and the data store used by the state management system, since–to the best of our knowledge–there is no middleware available to abstract out such dependency.

Figure 1 (top) illustrates the dependencies between the function processing logic and the API exposed by the state management system. Notice that changes in the NF requirements, new use cases, or upgrading the existing data store with new features, require a coding effort that significantly delays the NF deployment in production. That is, the process of identifying all the state variables in thousands of lines of code and adapt them to a new data store API is error-prone and time-consuming [2], [8].

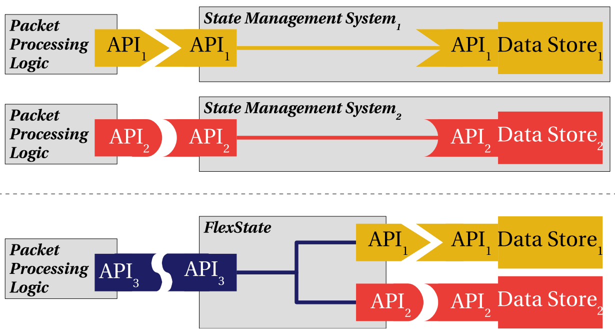


FIGURE 1. **FlexState middleware. The APIs exposed by state management systems are tightly coupled with the data store used internally (top). FlexState(bottom) provides an abstraction layer that exposes a single API to simplify data store changes and upgrades.**

Table 1 shows how different optimization goals influence the interface design and the data store selection. For instance, the state management system *CHC* [9] uses a custom key-value store to support *method call shipping*, by which the NF can offload some operations on the collections to the data store itself. Assume that a given network operator is using a state management system without any method call shipping support, like for example *Split/Merge* [1]. The expectation is that refactoring efforts should be limited to extending the NFs code to leverage the new feature. In practice, due to the heterogeneity of the APIs, NF developers also need to refactor code sections which reference the features that are common

TABLE 1. **Comparing state management APIs.**

		API Features					
System	**Goal**	Get/Set	Collections	Consistency Tuning	Locking	Timers	Merging
Split/Merge [1]	S	✓	○	✓	✓	✓	✓
OpenNF [2]	A	✓	○	✓	○	○	✓
StatelessNF [4]	R	✓	○	○	○	✓	○
S6 [5]	P	✓	✓	✓	○	○	✓
libVNF [12]	S	✓	○	○	○	○	○
CHC [9]	P	✓	✓	○	○	○	○

Each system is designed around a different goal: Accuracy (A), Performance (P), Reliability (R), or Scalability (S). The goal drives the data store selection, which ultimately affects the API of the state management system; ✓ indicates that the feature is supported, while ○ indicates that the feature is not supported. FlexState is designed to abstract out the heterogeneity of the APIs exposed by these state management systems. The API exposed by FlexState should be able to offer all the features supported by the APIs of the state management systems.

to the two data stores, e.g., *get/set calls*, in both previous and upgraded management systems.

The need for refactoring all API calls when adopting a new data store, even if such calls are semantically similar, represents a significant obstacle for network operators, which translates into data store vendor lock-in as well as incurs delays in developing network functions.

In this paper, we argue that adapting the code base to different data stores must not require continuous refactoring but can be provided through configuration. To this end, we propose FlexState, a state management system that introduces an abstraction layer for decoupling the NF state management from the adopted data store. As shown in Figure 1 (bottom), FlexStateenables NF developers to access and manipulate the state of NFs through a single API exposed to the packet processing logic while leveraging a range of data store drivers to translate API operations into data-store-specific query language. FlexStateis inspired by a multitude of successful projects that introduce abstraction layers to minimize development times. These are, for instance, *Apache Libcloud* [10] that implements a library for interacting with many cloud service providers through a unified API; or the *Serverless framework* [11] which offers an open source command line interface to deploy serverless applications across arbitrary platform providers.

In NF development, however, it is critical that introducing an abstraction layer does not affect the capability of NFs to process packets at line rate. Therefore, FlexStateis designed to make the best use of the resources available, e.g.,by scaling adequately. In Section V, we use two structurally different data stores to demonstrate that FlexStaterequires no modification on the NF's packet processing code while still preserving line-rate performance and scalability.

The rest of the paper is organized as follows. Section II provides background information and motivations behind FlexState, while Section III elaborates on its design and architecture. Implementation details are discussed in Section IV.

Evaluation results are presented in Section V, while we discuss the limitations of FlexStateand the directions for future work in Section VI. Finally we conclude in Section VII.

II. BACKGROUND AND MOTIVATION

Network functions, e.g.,NATs and load balancers, can be deployed in different scenarios, such as data center interconnects or enterprise networks, and for a variety of use-cases ranging from latency-sensitive to bandwidth-intensive. When deployed in production environments, NFs are expected to scale with the traffic load. For this reason, there can be multiple NF instances acting on a packet flow, or sharing states across multiple flows concurrently. Furthermore, within each NF instance, developers can also parallelize the packet processing to fully utilize the available CPU cores and increase performance. In this section, we describe the implications of the NF design choices mentioned above on the state management system.

A. NETWORK FUNCTION STATE

NFs are stateful entities that require timely data access and the ability to operate on the variables used to track the state of packet flows in the incoming traffic. The state of a NF can be arranged in two categories: *per-flow* state, and *cross-flow* state [9]. The per-flow category represents the state processing corresponding to packets of a specific flow. In contrast, the cross-flow state represents the state information considered when processing packets from all the flows traversing the NF. For instance, the per-flow state in a NAT NF contains the pair of IP addresses of a given TCP/UDP flow, while the cross-flow state includes the available IP addresses and port numbers that can be used for the translation.

B. STATE MANAGEMENT SYSTEMS

Maintaining the state information of NFs at line rate is challenging in terms of performance and consistency because each NF instance may have multiple threads processing packet flows, and there may be multiple NF instances in the network. For this reason, researchers have explored several state management alternatives to handle the entire life-cycle on behalf of the NFs by taking care of aspects such as consistency and correctness of data.

Table 1 summarizes some of these state management systems, each of which is optimized for a specific goal. Note that some systems are tailored for reliability, thus prioritizing that the NF state is always available, while other systems are designed for scalability, thus focusing on the capability of supporting a varying traffic load in an elastic, eventually consistent, manner. Developers can select the data store that is most suited for their use-case(s), according to the goal for which the system is developed. Some examples are key-value data stores for maximizing reliability goals [4]; or distributed implementations of hash tables for high performance objectives [5].

C. Problem Description

As mentioned in Section I, a common issue across all state management systems is that they are ultimately tightly coupled to a given data store, and do not provide any simple explicit mechanism for migrating to a different data store. Due to this coupling, different state management systems expose different APIs, even when the API calls are semantically equivalent, e.g., *get/set calls*. As a consequence, adopting a different data store requires changes in the API of the state management system, which entails invasive refactoring of the network function code while preserving its functionality.

EXAMPLE 1

The Bro intrusion detection system (IDS) [13] uses timers to handle cases in which the response of a DNS query is not received within certain time budget [14]. To manage the state of Bro, a network developer could initially adopt Split/Merge because it supports timers (Table 1). However, to improve the network reliability to deal with larger traffic volumes, the network developer must consider, for instance, using RAMCloud as proposed by state management systems tailored for reliability such as StatelessNF. Unfortunately, the RAMCloud API is not compatible with Split/Merge API, thus requiring refactoring the code for the new data store. In this example, the code of Bro using timers needs to be changed according to the RAMCloud API.

Another limitation, caused by tightly coupled data stores, is the inability to streamline–in the production NF–new features, bug fixes, and performance enhancements that are constantly released by the data store developers. Network operators would highly benefit from incorporating constant data store upgrades in their state management system, but this is a challenging process [8]. High refactoring costs usually result in some form of lock-in for network operators.

EXAMPLE 2

A network developer may implement the NFs with StatelessNF, which internally uses the RAMCloud data store as mentioned above. It is possible that a new RAMCloud version is released with major API changes. Note that such significant changes are not unusual, and are typically tracked by compatibility matrices [15], [16]. Unfortunately, StatelessNF currently does not have any built-in mechanism to support changes of the RAMCloud API, so the data store cannot be upgraded without incurring significant refactoring efforts.

D. Goal

The limitations mentioned above motivate our goal *of decoupling the NF state management from the data store*. As mentioned in Section I, we borrow the design principles from projects such as Apache Libcloud [10], the Serverless framework [11], and Prisma.io [17], which introduce an abstraction layer between coupled entities to minimize code refactoring. These approaches follow best practices in software engineering [18], and we leverage their architectural patterns to design FlexState, our NF state management system that allows to alternate across multiple data store APIs by simply modifying configuration parameters.

Notice, however, that while decoupling reduces code refactoring when upgrading existing data stores, or migrating to new ones, the additional level of indirection can affect performance. So, to constitute a compelling solution, FlexStatemust not affect the capability of NFs to process packets at line rate and, to do so, must be able to scale properly so that the load is distributed evenly across the available resources.
Therefore, the challenge addressed in this paper is two-fold.

- Enable flexibility in the choice of the data store used for NF state management
- Preserve scalability and line-rate performance.

Next, we provide more details on the FlexStatesystem design, architecture, and implementation.

III. FlexState ARCHITECTURE

In the following, we describe the key components of FlexState, namely the API and the data store drivers, and we explain how they can be used by NF developers and network operators (Section III-A). We then describe how FlexStatemanages state information (Section III-B) and the optimizations we designed to enable NFs using FlexStateto approach line rate (Section III-C).

A. KEY ENABLERS

1) API

The main goal in designing FlexStateAPI is to provide the features required by packet processing logic of NFs. To identify the features to be included, we use the classification of the APIs exposed by the other state management systems shown in Table 1. We present here how FlexStatesupports the two main features, namely the support for get/set operations and collections. We discuss how FlexStatecan support the remaining features in Section VI-B.

FlexStateAPI provides a set of data structures, each supporting a range of operations. Each data structure has a type, which determines the operations supported on the data structure. When a data structure is instantiated using the FlexStateAPI, the NF developer assigns an identifier (id) to it. Both the type and id are used to identify the data structure in the data store as explained in Section III-B2.

NAME-VALUE PAIRS AND COUNTERS

The NF developer can use a name-value pair to save a generic blob of data using a string as an identifier. The API calls exposed on such name-value pairs correspond to a Create, Read, Update, and Delete (CRUD) interface. Note that Read and Update calls correspond to Get and Set calls in key-value stores. In addition to name-value pairs, FlexStateexposes a dedicated data structure for counters. Indeed counters are used in a multitude of tasks, such as counting the total number of active flows, and they are natively supported by many data stores [19], [20]. In the FlexStateAPI, counters expose the same CRUD calls of the name-value pairs, and they also expose the call `add(value)`, which adds the specified `value` to the current value of the counter.

COLLECTIONS

Similarly to other state management systems [5], [9], FlexStateexposes collections, namely lists, sets, and maps. In addition to the CRUD interface, the calls exposed by collections take inspiration from the corresponding data structures in the C++ standard containers library [21]. FlexStatealso exposes countermaps, i.e.,maps whose values are counters. They are useful in many NF tasks, such as counting the number of packets for each active flow. Countermaps expose the same calls as a regular map, but they also expose the `addTo(key, value)` call, which adds the specified `value` to the current value of the counter identified by `key`.

2) DATA STORE DRIVERS

The goal of a data store driver is to translate FlexStateAPI calls using the query language of the data store. In this way, when changing data store, it is only needed to configure FlexStateto use the appropriate data store driver because the NF packet processing logic is written using the FlexStateAPI and it does not need to be changed. An example of translating API calls is shown in Table 2. A key challenge here is to realize a simple mechanism that allows network operators to change the driver adopted. In FlexState, the network operator compiles a configuration file, which is fed to the state management system. The network operator specifies the driver to be adopted and the parameters required to connect to the data store, i.e.,IP address and port. To specify the driver, the network operator uses a label, which identifies the data store of the driver.

When a new data store or a new major version of an existing data store is released, a driver can be written or upgraded to incorporate such a data store in FlexState. Developing the driver requires implementing FlexStateAPI calls only. Once the driver is developed, it can be integrated in FlexStateby providing a new label in the configuration file to identify the data store. After the integration process, network operators can set up the new data store by compiling the configuration file accordingly, and FlexStatewill start using the new integrated data store for state management.

B. STATE MANAGEMENT

1) ORGANIZATION

A state management system must ensure correctness of state variables, for example making sure that concurrent write operations do not corrupt the state information. While previous work have extensively discussed the difficulties in handling cross-flow state [5], [9], we argue that the handling of per-flow state is also not trivial. Indeed, the state management system must appropriately handle race conditions on state information when a NF instance runs on multiple cores.

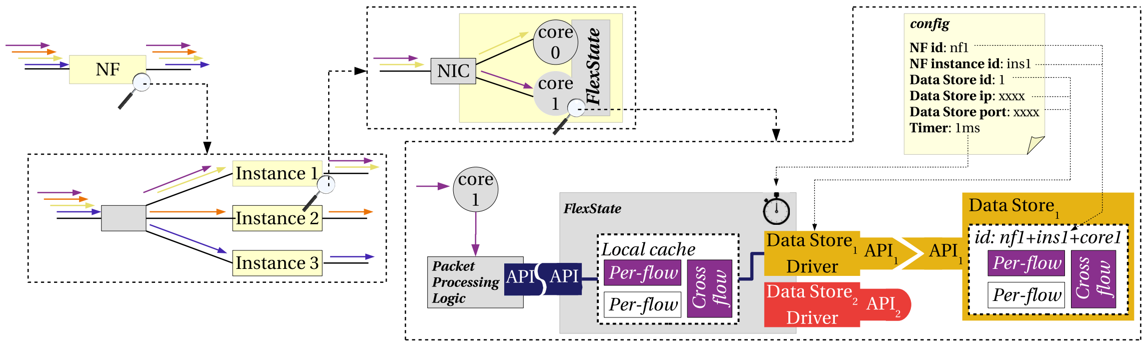


FIGURE 2. FlexState architecture. A network function can run over a variable number of instances, each of which processes a subset of flows. Within each instance, the network interface card (NIC) typically applies receive side scaling (RSS) to load-balance the flows across the cores available for each instance. On each core, the packet processing logic sends state operations to FlexState, which handles the state in a local cache. FlexStateuses a configuration file describing *a)* the data store driver to use and the information to reach the data store, and *b)* the frequency at which the state in the local cache is pushed to the data store.

TABLE 2. Examples for API conversion.

Type & Call	Redis	Cassandra
Counter add(n)	INCRBY nf1@ins1@1@Counter@counter_id n	UPDATE nf1@ins1@1.Counter SET value = value + n WHERE key=counter_id
Map insert(k,n)	HSET nf1@ins1@1@Map@map_id k n	INSERT INTO nf1@ins1@1.Map (key1, key2, value) VALUES (map_id, k, n)
Countermap addTo(k,n)	HINCRBY nf1@ins1@1@Countermap@cmap_id k n	UPDATE nf1@ins1@1.Countermap SET value = value + n WHERE key1=cmap_id AND key2=k

The data store drivers use the information provided by FlexState as described in Section III-B2. NF id, NF instance id, and core id are taken from Figure 2. Symbol @ is used to separate the fields.

FlexStatesolves this problem using *partitioning*. According to this principle, data is partitioned among a group of executors, such as NF instances or cores, and each executor accesses and modifies only its own data. In this way, the executors are made independent from each other and they do not incur race conditions because there is no shared data. When considering multiple NF instances, FlexStatedivides the state information among the instances, and each instance accesses and modifies its own state information only. As an example, when considering a cross-flow state information, such as a counter for the total number of flows traversing the NF, the counter is split into a set of independent counters, each one of them associated to a single instance.

Partitioning is not applied only across different NF instances, but also within each instance. Figure 2 illustrates how FlexStateapplies partitioning within a single NF instance. FlexStateleverages the fact that modern Network Interface Cards (NICs) support Receiving Side Scaling (RSS). When RSS is activated, the flows arriving at the NIC are distributed evenly among the cores made available to the NIC. Crucially, the NIC forwards packets of the same flow always to the same core [22]. As a consequence, for each flow, there is a single core processing its packets, so race conditions cannot occur handling per-flow state. When considering cross-flow state instead, FlexStateapplies partitioning by splitting the cross-flow state among the cores allocated for the NF instance. Considering the example of the counter for the total number of flows, FlexStatesplits the NF instance counter into a set of independent counters, each one of them associated to a single core. While each counter is still in the cross-flow state, it is accessed and modified only by a single core, and thus race conditions cannot occur.

Note that designing partitioning-aware NFs is a non-trivial task. NF developers need to split the state information across cores and NF instances that are typically shared. In Section V-A4 we provide two examples of how to perform this splitting. Moreover, network operators need additional tools to view the NF state as a single entity, e.g.,to examine the overall load across all NF instances. In Section VI-B we discuss the need for merging functions, also called combiners [1], which are used to obtain a single representation of state that is scattered across NF instances.

2) IDENTIFICATION

Each core of each NF instance manages a piece of NF state in an exclusive fashion. The NF state is stored in a data store which is shared by all cores of the NF instance. Moreover, the data store might be shared also by other NF instances and by other NFs. Therefore, there is a need for creating unique identifiers for state information so that partitioning can be applied in the data store.

In FlexState's configuration file, the network operator specifies two additional pieces of information, a) a NF identifier, and b) a NF instance identifier. These pieces of information are used to distinguish data of different NFs and to distinguish data of different instances of the same NF, respectively. FlexStatealso leverages the id of the core from which state operations are being issued to distinguish data used by different cores of the same NF instance. For each data structure created using the API, FlexStatecreates a unique key combining a) the NF identifier, b) the NF instance identifier, and c) the id of the core. FlexStatealso combines the type of the data structure performing the state operation, e.g.,counter, and the id of the data structure assigned by the NF developer. These two pieces of information allow distinguishing data managed by the same core. An example of how a unique id is created can be found in Appendix A.

C. PERFORMANCE OPTIMIZATIONS

1) *no_wait* CALLS

All API calls described so far return either a result of a query, e.g.,the data corresponding to a *get* call, or an acknowledgement of completed operation. While normally the NF waits for the response from the data store, there are situations in which waiting for the response is not desirable. For example, if state operations are issued in the packet processing loop, waiting for responses from the data store can slow down the NF. Some state management systems address this issue by adopting *no_wait* calls, which issue state operations without waiting for a response from the data store [5], [9]. Therefore, we complement the regular API calls with no_wait calls that can be used in the packet processing loop without slowing down the NF. Note that not all calls can have a no_wait version. For example, if a NF uses the *get* call to obtain data from the data store, then it needs to wait for the response from the data store. In our experience, normal calls are used only in initialization or shutdown of the NF, but not within the packet processing loop.

2) ASYNCHRONOUS UPDATES

The rate at which NFs process packets can be very different from the throughput of data stores, i.e.,number of operations per second [23]. In this case, the overall processing rate of the system corresponds to the rate of the slowest between the data store and the packet processing logic. The problem stems from the idea of performing state operations on the data store every time a packet is processed in a synchronous fashion. FlexStatesolves this problem by decoupling the packet processing loop from the state management operations. The packet processing loop operates on a local cache of the state, thus avoiding the need to communicate synchronously with the data store. A periodic operation is then issued to update the state on the data store with the changes that have been performed on the local cache. In effect, this corresponds to asynchronous updates to the data store.

A key aspect to consider is the frequency at which the update operations are carried out. Depending on the use-case, a network operator might require high availability, and thus to have very frequent updates on the data store [22], [24]. FlexStateallows the network operator to configure the frequency of updates to the data store in the configuration file, as shown in Figure 2. More specifically, the network operator sets the time gap between updates to the data store, e.g.,1 ms. By decreasing the value, the network operator increases the frequency of updates to the data store at the expenses of a higher amount of traffic between FlexStateand the data store.

IV. IMPLEMENTATION

Our FlexStateprototype consists of approximately 5K lines of C++ code. In the following, we describe the tools and techniques we used to implement each component of FlexState.

A. KEY ENABLERS

The key goal of FlexStateis to enable changing the data store driver without requiring changes in the code of the packet processing logic of the NFs nor in the state management system. Therefore, we implemented FlexStateAPI as an *interface* that is instantiated by the data store drivers. FlexStateinternally uses this interface to issue state operations, thus remain agnostic to the data store driver being used.

To exemplify how FlexStateAPI can be used with different data stores, we implemented the drivers for a range of data stores in our FlexStateprototype. We choose Redis [19] and Cassandra [20] because they belong to different data store families, i.e.,Redis is a key-value store with a flat key space while Cassandra organizes data in tables. Redis has been used in NF systems due to its consistency guarantees [12], while the fault tolerance capabilities of Cassandra can be leveraged with use-cases with very stringent availability requirements [25]. Moreover, Redis and Cassandra are both carrier-grade data stores, i.e.,they are used and maintained by major IT companies: using carrier-grade data stores for NF state management provides further benefits, as discussed in Section VI-C. We also implemented the driver for an in-memory hashmap. The hashmap is not shared among NF instances and it executes locally to each NF instance, i.e.,it runs in the same host of the NF instance. We use the hashmap only for benchmarking purposes.

Table 2 shows a few examples of how the data structures and the API calls of FlexStateare converted by the data stores drivers. Supporting counters is straightforward because both Redis and Cassandra natively support counters, and the `add(value)` call of the FlexStateAPI can be mapped directly to the corresponding calls in Redis and Cassandra, respectively `INCRBY` and addition operand. Supporting maps and countermaps in Redis is easy as well because the data store supports both data structures and thus it natively exposes calls for inserting an element into a map and incrementing a value in a countermap. With Cassandra, we implement maps and countermaps by expanding them in normal tables because the native support for maps in Cassandra is inefficient. We discuss this aspect in more detail in Section IV-B.

B. STATE MANAGEMENT

FlexStateleverages RSS to distribute the flows across the available cores and partitioning to avoid inter-core contention, and thus improve the performance and the scalability of the system. In our implementation, we use Seastar [26], a framework that has been used successfully in other related work [14]. Seastar distributes the flows across the available cores by configuring the NIC to apply RSS and linking each hardware queue of the NIC to a different core. If the number of available cores is higher than the number of hardware queues in the NIC, then Seastar creates software queues for the remaining cores and it performs RSS in software to distribute the flows evenly among all available cores. For each available core, Seastar creates a thread, it pins the thread to the core, and it configures the thread to process the packets of the queue linked to the core. Seastar also facilitates partitioning by creating per-core data structures, which are accessed and modified only by the thread assigned to the core. Lastly, Seastar natively integrates with DPDK [27], which we adopt to improve the performance of the system.

Data store drivers fetch and organize the state information in the data store leveraging the unique keys created by FlexStateas described in Section III-B2. Each data store has a specific way of organizing data: Cassandra, for example, organizes data in tables which can be grouped in different key spaces, while Redis has typically a single flat key space. In our implementation, the data store driver for Redis uses the keys of FlexStatedirectly to store and fetch state information. For example, the key `nf1@ins1@1@Counter@abc` is used as-is to identify the counter `abc` used by core 1 of NF instance `ins1` of NF `nf1`. Instead, the data store driver for Cassandra uses first the NF id, the NF instance id, and the core id to identify a key space. Then, data structures of different types are stored in different tables, and the data structure id is used to identify the data structure within a table. Using the previous example, the key space identifier is `nf1@ins1@1`, the table is `Counter`, and the id of the data structure is `abc`. To fetch the value of the counter, we use the query `SELECT value FROM nf1@ins1@1.Counter WHERE key=abc`.

Using verbose queries and receiving bulky replies can quickly saturate the link between FlexStateand the data store, ultimately decreasing the performance of the system. For this reason, data store drivers must use the data structures offered by the data store in the most efficient way. For example, the data store driver for Redis directly uses collections and their calls exposed by the data store. Cassandra also supports collections, but they expose a limited number of calls, e.g.,it is not possible to fetch a single element from a map in an efficient manner. For this reason, the data store driver for Cassandra implements maps by expanding them in tables, and it uses queries on tables to perform operations on maps efficiently. As shown in Table 2, every key space in Cassandra has a table `Map`, which contains maps. For each map, the table `Map` contains the id of the map (in column key1) and all the key-value pairs of the map (in column key2 and column value, respectively).

C. PERFORMANCE OPTIMIZATIONS

To decouple packet processing logic and state management, we cannot schedule the state management operations on the threads used for processing packets. For each Seastar thread, we create a dedicated thread to perform the state management operations. Periodically, the Seastar thread schedules the state updates for the data store to its state management thread. The network operator uses the configuration file to set the frequency with which the Seastar threads schedule state updates. To implement the state managements threads, we use libevent [28] because it integrates easily with the libraries for communicating with the data store, i.e.,hiredis-vip [29] for Redis and DataStax C++ Driver [30] for Cassandra.

V. EVALUATION

The design of FlexStateguarantees the achievement of our first objective, i.e.,FlexStateoffers the ability to change the data store adopted for NF state management by means of configuration. Nevertheless, we need numerical evidence that the second objective is achieved, i.e.,FlexStateis able to scale with the assigned resources, and it allows the NFs to process packets approaching line rate. In particular, attention should be paid to all the aspects that might influence line-rate performance, such as the hardware setup and the location of the data store. For these reasons, the aim of our evaluation is to answer the following questions.

DOES OUR TESTBED SUPPORT LINE RATE?

Answering this question is important because we want to make sure that the testbed we use for evaluating FlexStateis able to serve packets arriving at line rate.

DOES FlexStateENABLE NFs TO APPROACH LINE RATE?

The goal of FlexStateis to provide flexibility in changing the data store without hampering performance. NFs using FlexStatemust therefore be able to process packets approaching line rate.

DOES FlexStateSCALE WITH THE NUMBER OF CORES MADE AVAILABLE TO THE NF?

NFs parallelize their packet processing across the cores made available to it, and FlexStatemust be able to support this and fully utilize the available resources.

HOW TO QUANTIFY THE BENEFITS OF PERFORMANCE OPTIMIZATIONS?

We believe no_wait calls and asynchronous updates to be crucial in enabling the NFs to approach line rate. To confirm our expectations, we need numerical evidence that shows the benefits of the performance optimizations in FlexState.

HOW DO THE DATA STORE AND ITS LOCATION AFFECT FlexState?

The decoupling between the state management and the data store, together with the proposed performance optimizations, should ensure that the location of the data store does not affect the system. For example, we want to verify that running the data store on the same node where FlexStateis running, i.e.,locally, or on another node, i.e.,remotely, does not affect the performance.

In the following, we describe our testbed (Section V-A) and the experiments we perform to answer our questions (Section V-B).

A. TESTBED DESCRIPTION

1) OVERVIEW

Our testbed is described in Figure 3. It comprises two Dell C6320 nodes [31], i.e.,node1 and node2. Both nodes are equipped with a Intel 82599ES 10GbE dual-port SFP+ NIC, which features two ports, port1 and port2. The nodes are connected with an 10 Gbps link for each port pair, i.e.,port1 and port2 of the first node are connected to port1 and port 2 of the second node respectively. The traffic is generated on node1 using Pktgen [32], a tool of the DPDK suite which is capable of saturating the 10 Gbps link. The other node, node2, is used for running the NFs atop FlexState, which in turn uses Seastar and DPDK to receive and send the packets. The first node, node1, is then used to collect the traffic processed by the NFs. The two nodes use port1 to exchange data traffic. If the data store is run locally, then the communication between FlexState and the data store occurs through the loopback interface; otherwise, the communication occurs through port2.

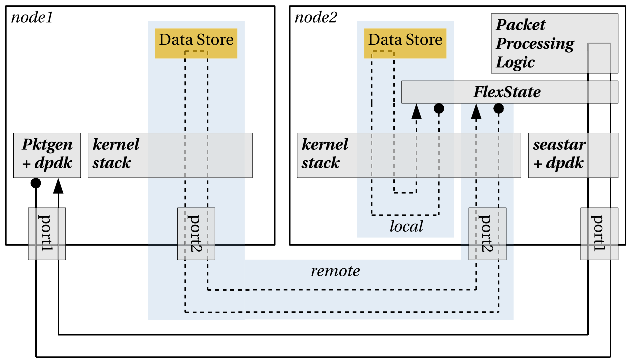


FIGURE 3. Testbed workflow. Pktgen sends data packets from node1 to node2 using port1, and FlexStateprocesses the data packets before sending them back. Simultaneously, FlexStatesends state updates to the data store, which can be local or remote. In the local case, the communication is confined to node2; in the remote case, FlexStatecommunicates with the data store on node1 using port2.

2) TRAFFIC GENERATION

We perform our experiments replicating a worst-case scenario, i.e., a scenario where the NF has to serve a large number of packets arriving at the line rate. We therefore generate packets of 64 bytes, which corresponds to the minimum size for an Ethernet frame [33]. The source and destination MAC addresses of the packets are set to the MAC address of the data interface of node1 and node2 respectively; the source IP address, destination IP address, source port, and destination port are generated randomly. The generated packets are stored into a pcap file, which is then used by Pktgen as input. The pcap file contains 50K different packets, which are sent over and over to the NFs for the whole duration of the experiment, resulting in 50K flows traversing the NFs. In each experiment, we configured Pktgen to stream the traffic for 15 seconds. To improve confidence in the results, we repeat our experiments using 10 different pcap files, and the results presented here are obtained by computing the average of the results over all experiments.

3) CONFIGURING NIC AND CORES

To test the scalability of FlexState, we run our experiments assigning to FlexStatea varying number of CPU cores. Each node is equipped with 48 cores (24 physical cores and 24 virtual cores). More specifically, each node consists of two NUMA nodes, NUMA1 and NUMA2. Each NUMA node contains 12 physical cores, and for each physical core there is an additional virtual core due to hyperthreading. In both nodes of our testbed, port1 has 16 queues [34]. Each queue is assigned to a dedicated core that processes the packets arriving in that queue [22].

We design and adopted a set of rules for deciding how to connect the available cores and the queues of port1. We took into consideration the DPDK guidelines [35] that recommend selecting distinct cores of the same NUMA node to which the NIC is connected, i.e.,NUMA2. For this reason, we also configured the physical cores of NUMA2 with `isolcpus`, `nohz_full`, and `rcu_nocbs` kernel flags. We order the 48 cores in the following manner: the twelve physical cores of NUMA2, followed by the twelve physical cores of NUMA1, the twelve virtual cores of NUMA2, and the twelve virtual cores of NUMA1; an experiment requiring n cores, selects the first n cores in this list. Note that Seastar creates software queues for the remaining available cores when the number of available cores is higher than the number of queues of the NIC port (Section IV-B).

4) NETWORK FUNCTIONS

We consider the following NFs in our experiments.

TESTPMD

We use testpmd [36] to assess the capabilities of the testbed and to obtain a baseline for comparing FlexState's performance. This tool of the DPDK suite performs simple operations on the packets, such as changing header information and forwarding, and it provides statistics about received, dropped, and transmitted packets. We run testpmd on node2 by connecting it directly to port1 through DPDK, and thus skipping the software layers of Seastar and FlexState. We configured testpmd to send back the received packets by swapping the MAC addresses. Unlike Seastar, testpmd cannot create additional software queues, so we run testpmd using up to 16 cores only.5

COUNTER NF

To measure the impact of the software layers of Seastar and FlexState, we implement a NF which just counts the packets flowing through it. To also measure the benefits of the asynchronous updates, we develop two versions of this NF. In the first version, for each received packet, the NF immediately updates the counter in the data store (Sync Counter). The second version uses asynchronous updates and FlexStateupdates the counter in the data store every 1 ms (Async Counter).

NAT AND LOAD BALANCER

To measure the performance of FlexStatewith regular NFs, we implement a NAT and a load balancer using the scaffolding provided by Kablan *et al.* [4]. The NAT substitutes the source IP and the source port of an incoming packet with a (IP, port) pair taken from a pool of available (IP, port) pairs. Each flow is assigned its own pair, i.e.,all packets of the flow are modified using the same (IP, port) pair. The load balancer distributes the incoming flows evenly among the servers in a given list. When a new flow arrives to the NF, the least loaded server is selected to serve the flow. We adopted partitioning to implement the two NFs. In our NAT we split the pool of available (IP, port) pairs into chunks and we assign a chunk to each core of the NF instance, while in our load balancer each core has its own load counters. Moreover, both NFs make use of no_wait calls and asynchronous updates, and FlexStatesends state updates to the data store every 1 ms. Note that, despite the logic of the load balancer is applied to received packets, all packets are eventually forwarded to node1.

B. EXPERIMENTS AND RESULTS

DOES THE TESTBED SUPPORT LINE RATE?

We configure Pktgen on node1 to send traffic to node2 saturating the 10 Gbps link, i.e.,14.88 Mpps [14]. We run testpmd on node2 and we configure it to send the received traffic back to node1. We vary the number of cores assigned to testpmd, and this internally determines the number of NIC queues used. Figure 4 shows the transmission rate of testpmd, i.e.,the number of packets forwarded back to node1 per second. testpmd is indeed able to transmit packets back at the same rate of reception, so we can conclude that the testbed supports line-rate communication. Note that increasing the number of

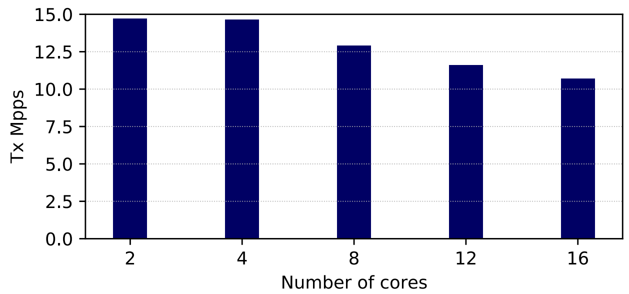


FIGURE 4. **Performance of testpmd. The testbed supports line-rate speed, i.e.,14.88 Million packets per second (Mpps).**

assigned cores determines a deterioration in performance due to the overhead in managing additional queues [37].

DOES FlexStateENABLE NFs TO APPROACH LINE RATE?

Figure 5 and Figure 6 show the performance recorded in our testbed by NAT and load balancer respectively. In particular, we measure the performance running each NF with all data stores, i.e.,hashmap, Redis, and Cassandra, and considering all the locations, i.e.,local and remote. We can see that the NFs both record a transmission rate of about 10 Mpps when we allocate 24 cores to FlexState. These results are in line with the performance recorded by the NFs using other state management systems [4], [14]. We can conclude that FlexState, similarly to existing state management solutions, allows NFs to approach line-rate packet processing.

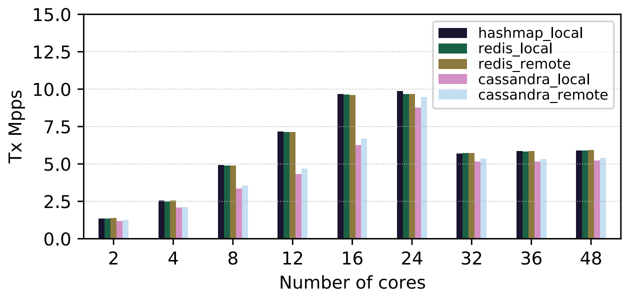


FIGURE 5. **Line-rate performance and scalability of FlexStatefor a NAT. FlexStateapproaches line-rate performance, i.e.,close to 10 Mpps, and it scales with the number of cores assigned. The performance drops as soon as FlexStateuses virtual cores, thus indicating that hyperthreading is not beneficial.**

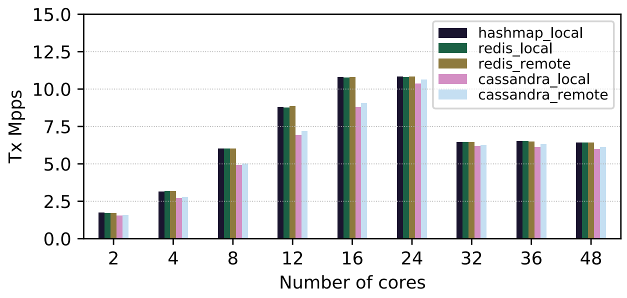


FIGURE 6. **Line-rate performance and scalability of FlexStatefor a Load Balancer. The legend is the same as in Figure 5. As in the case of for the NAT, FlexStateapproaches line-rate performance, i.e.,close to 10 Mpps, and it scales with the number of cores assigned. The performance drops as soon as FlexStateuses virtual cores, thus indicating that hyperthreading is not beneficial.**

DOES FlexStateSCALE WITH THE NUMBER OF CORES MADE AVAILABLE TO THE NF?

Figure 5 and Figure 6 show how the NFs perform when we vary the number of cores assigned to FlexState. In both cases, there is a steady increase in performance when going from 2 to 24 cores, which highlights the capability of FlexStateto scale with the resources available. Nevertheless, when we assign to FlexStatemore than 24 cores, we can see that both NFs record a drop in their performance. We believe that this is due to the usage of virtual cores. FlexState uses the physical cores of node2 when it is assigned up to 24 cores, and it starts using the virtual cores when it is assigned more than 24 cores. We have detailed the allocation of CPU cores in Section V-A3. When more than 24 cores are assigned, the physical cores have no idle time because they are busy in processing packets. Consequently, using virtual cores forces interleaving between non-idle cores, which worsens the performance.

One can note that the behaviour of testpmd is very different from the one of the two NFs, i.e.,testmpd performance worsens when increasing the number of cores. We suspect that this depends on the overhead of the packet processing logic. testpmd simply performs a swap of the MAC addresses, while NAT and load balancer operate on several data structures and change several header fields before sending the packet out. The main overhead for testpmd is therefore distributing the flows of packets to a high number of queues. Instead, NAT and load balancer benefit from distributing the packet flows to a high number of cores because their main overhead is due to their own packet processing logic.

HOW TO QUANTIFY THE BENEFITS OF PERFORMANCE OPTIMIZATIONS?

We compare the performance of Sync Counter, which does not use no_wait calls and which communicates synchronously with the data store, with the performance of Async Counter, which uses asynchronous updates instead. We show the comparison in Figure 7. We can see that Async Counter outperforms Sync Counter; more specifically, Async Counter reaches around 12 Mpps, while Sync Counter is never able to record more than 2 Mpps.

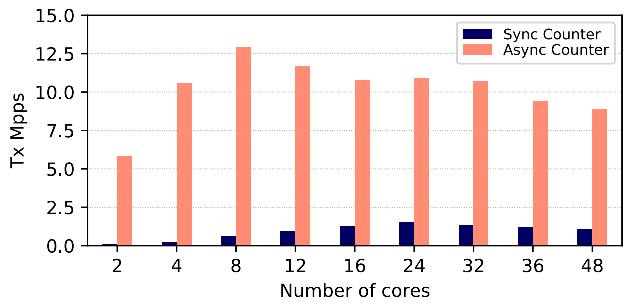


FIGURE 7. **Sync Counter vs Async Counter. Async Counter outperforms Sync Counter in terms of transmitted packets per second.**

We can observe that Async Counter performs best when we assign 8 cores to FlexState, which confirms the need to find a trade-off between the overhead of the packet processing logic and the overhead of distributing the flows to a higher number of queues. The packet processing logic of Async Counter only increases a counter, in addition to swapping the MAC addresses to send the packet back; the overhead of its logic is higher than the one of testpmd, but smaller than the one of NAT and load balancer. They just have to increase a counter, but it is still higher than the one of testpmd. As a result, assigning up to 8 cores benefits the performance, while the overhead of managing additional queues becomes too high when assigning more than 8 cores.

HOW DO THE DATA STORE AND ITS LOCATION AFFECT FlexState?

Given a data store and its location, we selected the number of cores which resulted in the NF having the best performance, and we reported the corresponding value in Figure 8. We can see that a) all NFs perform close to line-rate performance, and b) for each NF, the difference in performance across different data stores and different locations of the data stores is negligible. These results confirm that FlexStateindeed allows the packet processing logic to operate at its own speed regardless of the data store being adopted.

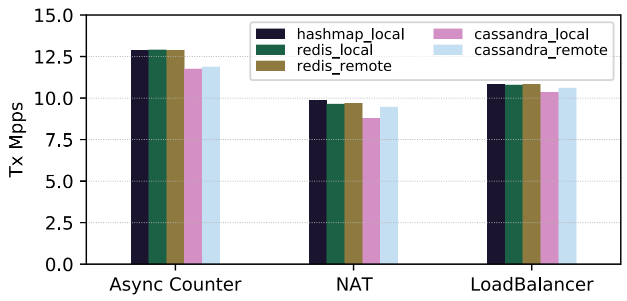


FIGURE 8. **Benefits of Asynchronous Updates. For each NF, the values are obtained using the number of cores that achieve the best performance for the data store_location pair. There are no substantial differences in performance across different combinations of data store and location.**

VI. DISCUSSION

We have shown that FlexStatecan abstract the data store used for state management without the need of major refactoring while allowing NFs to approach line-rate performance. However, the choice of applying partitioning to manage NF state may limit the scenarios in which FlexStatecan be used. In Section VI-A we discuss these limitations and how we plan to address them. Moreover, we have discussed how FlexStatesupports name-value pairs and collections. However, these features correspond to only two of the features we have identified in Table 1. So, in Section VI-B we elaborate on how FlexStatecan support the remaining features. For completeness, we finally discuss our future work in Section VI-C.

A. LIMITS OF PARTITIONING

We identified two major issues that may arise when applying partitioning to manage the state of NFs.

The first issue is that the thresholds originally imposed on the entire state of the NF, such as the maximum size of a table, are enforced on a per-NF-instance, per-core level with partitioning, and consequently monitoring the status of the NF requires additional maintenance. For example, the pool of (IP address, port) pairs to masquerade new flows in a NAT is divided among the NF instances, as well as among the cores of each NF instance, so only a limited number of pairs is available for each core. A temporary, uneven distribution of the traffic among the cores of a NAT instance may result in one core running out of (IP address, port) pairs despite the NAT still having spare pairs distributed over other cores.

The second issue is that NFs may require to know in advance the number of NF instances and the number of cores of each NF instance to prepare the data store properly. Considering the NAT example again, the pool of (IP address, port) pairs must be divided according to the pre-planned number of NF instances and cores *before* the NAT is run. This requirement limits the flexibility of the system because the network operator must know the maximum number of instances that a NF will require in advance, as well as the number of cores of the physical nodes running the NF instances.

Both issues highlight a need for complementing FlexStatewith flexible mechanisms to reorganize the state information in the data store, for example, according to run time events such as thresholds being hit or new NF instances joining. More generally, these mechanisms fall into the group of techniques that researches have been developing to enable more flexibility in partitioning, for example by configuring the forwarding of the incoming requests to the cores based on the size of the request [38]. In the NAT example, there is a need for redistributing the (IP address, port) pairs when one core exhausts the pairs allocated to it. Defining the complete list of run time events such mechanisms should react to, and making sure that their actions do not corrupt the local cache of state information, requires further investigation.

B. CONSIDERATIONS ABOUT OTHER FEATURES

CONSISTENCY TUNING

Some state management systems support consistency tuning, i.e.,the NF developer can set the consistency of the state information using the API [5]. FlexStateremoves the need for this feature by applying partitioning: state information is not shared between executors, i.e.,cores or NF instances, so the state information is always up-to-date for each executor.

LOCKS

Despite the advantages of partitioning, certain networking tasks may require sharing the state among several cores or even several instances [39]. For example, an IDS based on Finite State Machine (FSM) models will require maintaining the state information about a group of flows [40], but such flows are not guaranteed to be processed by the same core. In this case, the FlexStateAPI can be extended to support two additional calls, `acquireLock` and `releaseLock`, which are translated by the drivers into data-store-specific mechanisms for acquiring and releasing locks respectively. For example, both Redis and Cassandra can make use of the `IF NOT EXIST` clause to mimic a lock acquisition. A NF could use such calls on any piece of state information to globally grant exclusive access to the state information. Nevertheless, making use of a locking mechanism in a distributed scenario is known to be a performance killer [9]. Our recommendation is to use locks only as a last resort.

TIMERS

Some NFs use timers to carry out their tasks. For example, when a new flow arrives, a malware detector performs a query to a registry for malware signatures, and it arms a timer to be able to react in case no reply is provided [14]. For this reason, some state management systems offer explicit support for storing timers and notifying the NF in case of expiration [1], [4]. FlexStatecan be extended to support timers by leveraging the Time-To-Live (TTL) property offered by data stores. FlexStateassociates a TTL to a record in the data store, and the NF arms the timer locally. In this way, even in case of failure of the instance handling the timer, a newly launched instance can query the data store for the timer record. In case the record is not in the data store anymore, then the timer has expired. Otherwise, the current TTL of the record can be used to arm a timer locally again.

MERGING FUNCTIONS

Partitioning makes it more difficult to have a comprehensive view of the status of the network function. For instance, when considering a load balancer, there is no single information representing the total load of the servers. Nevertheless, this drawback can be mitigated by introducing merging functions, i.e.,functions that merge scattered data to provide unitary information [1], [2], [5], [41]. Considering the example of the load balancer, a merging function retrieves the load of the servers for each core of each instance of the load balancer and sums them together, thus providing the user with a unitary value. Network operators can obtain a comprehensive view of the status of the network by running these functions in a cyclic fashion.

C. FURTHER EVALUATION AND FUTURE WORK

MULTIPLE NF INSTANCES

Our evaluation shows that FlexStateis able to scale with the number of cores assigned to it. Note that FlexStateis designed to scale with the number of instances as well. Indeed, in the same way cores of a network function instance are not required to synchronize with each other, different instances of the same NF are not required to communicate with each other. Network operators only have to assign a unique id to each instance by writing it in the configuration file. We plan to evaluate the scalability across several instances as future work.

CARRIER-GRADE DATA STORES

A key advantage of FlexStateis that any data store can be adopted for state management provided that the driver for it has been developed. Instead of using ad-hoc data stores, network operators can use carrier-grade data stores for managing the state of their network functions. Carrier-grade data stores offer several advantages in terms of cost, maintainability and support given their typically large community of users. While they might not be suitable for all use cases, we believe network operators can benefit from this possibility in specific contexts. Inspired by similar projects [42], we aim to complement FlexStatewith a range of drivers for the most widely used data stores.

IMPROVING EFFICIENCY

Use cases such as Augmented Reality (AR) require both high performance and high availability [7], [43]. While we have shown that high performance can be achieved by decoupling the packet processing loop from state management, high availability can be approached by increasing the frequency of the updates to the data store and mirroring the data store in multiple locations. In its current stage, FlexStatepushes the updates to the data store without performing any optimization. For example, FlexStateinserts new entries in a map one at a time, while inserting multiple entries using a single query would reduce the volume of traffic towards the data store. We are planning to complement FlexStatewith techniques that allow representing state changes in an efficient way; for instance, the approach of Nobach *et al.* [44].

D. Other Related Works

ABSTRACTION LAYERS

In recent years, we have witnessed how researchers have solved key problems in computer science by leveraging abstractions [18]. Focusing on the *networking* domain, the most glaring example is the introduction of OpenFlow [45], which abstracts out the details of the network equipment while providing a simple interface to network administrators. Similarly, the FlexStateAPI abstracts out the details of a single data store and provides a unified interface by which NF developers can write the packet processing logic. The FlexStateAPI resembles a Database Abstraction Layer (DBAL), a well-known and mature concept in software engineering [46]. Researchers and software developers have proposed several DBALs throughout the years [47], [48] but, to the best of our knowledge, none of them provide the features required to manage the NF state as described in Table 1.

CONCURRENCY

Reducing contentions between threads is critical to the performance of networked system [49]. FlexStateleverages partitioning to essentially eliminate the communication between different threads; still, there are instances in which applying a partitioning model is infeasible because of the need for having data shared between threads (cf. Section VI). An alternative is to apply a different concurrency model, e.g.,the actor model [50], according to which the operations on a data structure are executed in strict sequence. This guarantees that data corruption cannot occur, regardless of the thread that actually carries out the operations. NFVActor [51] is a system for managing NFs that leverages the actor model, nevertheless the system has limited support for shared state between NF instances.

OTHER NETWORK LAYERS

State management is a thorny problem at every layer of the network stack. While FlexStatedeals mostly with L3/L4 NFs, the problem appears in both lower, e.g.,L2, and upper layers in the stack, e.g.,the application layer, although the requirements are more homogeneous in these cases. Systems such as EP2 [52] and SNAP [53] focus on relieving the NF developer from the burden of handling NF state while maintaining high performance by keeping state locally to the NFs. Instead, availability is more relevant than performance at the application layer. This led to the spread of data-centric paradigms, such as serverless computing, according to which applications have no own state but state is stored in a remote data store [54], [55]. An example of such serverless systems is Conductor from Netflix [56].

VII. CONCLUSION

The systems that manage the state of network functions (NFs) are tightly coupled to the specific data store implemented internally, so NFs relying on the APIs of such systems are also tied to the specific data store. As a consequence, network operators cannot easily upgrade data stores or adopt new ones that are more suitable for specific use-cases or customer requirements without incurring high refactoring costs. Inspired by projects in the cloud domain, we propose an abstraction layer that decouples NF state management from the underlying data store, in a way that the data store adoption becomes a *configuration parameter*. FlexStateimplements this abstraction layer while scaling with the resources available to minimize the performance costs of decoupling. Our experiments show that FlexStateenables NFs to process packets approaching the line rate. While further efforts are required to address the limitations identified in Section VI, we believe that the results of our research represents a significant step in bringing flexibility to network function development.

.

APPENDIX A STATE IDENTIFICATION EXAMPLE

We write the packet processing logic of a simple NF which counts the total number of packets going through it. The pseudocode of the packet processing logic using FlexStateAPI follows:

```
// create the counter
Counter pktCounter =
    VariableFactory::createCounter("pktCounter");

// called for every received packet
void processPacket(packet) {
    // increase the counter value
    pktCounter.add(1);
}
```

Note that the NF developer has assigned the id *pktCounter* to the variable. The id also corresponds to the name the variable has in the code, but this is not required by FlexState.

A network operator who wants to run the NF compiles the following information in the configuration file fed to FlexState:

```
[...]
NF id: nf1;
NF instance id: ins1;
[...]
```

The network operator runs FlexState, which initiates the NF instance. The network operator assigns a number of cores to FlexState. Let us focus on core 0. FlexStateneeds to uniquely identify the counter being used by core 0 in the data store. The key that FlexStatebuilds combines the information as shown:

```
nf1 + ins1 + core0 + Counter + pktCounter
```

The data store driver uses the information in the key to identify the counter within the data store (Section IV-B). The presented schema for identifying state information allows distinguishing a) state information of different NFs, b) state information of different NF instances, c) state information of different cores in the same NF instance, and d) different data structures that are labeled with the same id.

APPENDIX B AVAILABILITY

The source code of FlexStateis available at the following URL: https://version.helsinki.fi/matteo.pozza/flexstate.

REFERENCES

[1] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. NSDI*, Lombard, IL, USA, Apr. 2013, pp. 227–240.

[2] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. and Akella, "OpenNF: Enabling innovation in network function control," in *Proc. SIGCOMM*, Chicago, IL, USA, Aug. 2014, pp. 163–174.

[3] (2020). *The Promise of CNFs*. Accessed: May 2020. [Online]. Available: https://www.redhat.com/en/blog/cnf-and-vnf-certification-red-hat-and-intel/

[4] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. NSDI*, Boston, MA, USA, Mar. 2017, pp. 97–112.

[5] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. NSDI*, Renton, WA, USA, Apr. 2018, pp. 299–312.

[6] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *Proc. SIGCOMM*, Los Angeles, CA, USA, Aug. 2017, pp. 43–56.

[7] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Commun. Mag.*, vol. 53, no. 2, pp. 90–97, Feb. 2015.

[8] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using statealyzr," in *Proc. NSDI*, Santa Clara, CA, USA, Mar. 2016, pp. 239–253.

[9] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *Proc. NSDI*, Boston, MA, USA, Feb. 2019, pp. 501–516.

[10] (2020). *Apache Libcloud. One Interface to Rule Them All*. Accessed: May 2020. [Online]. Available: https://libcloud.apache.org/

[11] (2020). *The Serverless Framework*. Accessed: May 2020. [Online]. Available: https://serverless.com/

[12] P. Naik, A. Kanase, T. Patel, and M. Vutukuru, "libVNF: Building virtual network functions made easy," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, Carlsbad, CA, USA, Oct. 2018, pp. 212–224.

[13] (2020). *An Open Source Network Security Monitoring Tool*. Accessed: May 2020. [Online]. Available: https://zeek.org/

[14] J. Duan, X. Yi, J. Wang, C. Wu, and F. Le, "NetStar: A future/promise framework for asynchronous network functions," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 600–612, Mar. 2019.

[15] (2019). *Cassandra Query Language (CQL) V3.4.0*. Accessed: May 2020. [Online]. Available: https://cassandra.apache.org/doc/old/CQL-3.0.html

[16] (2020). *Compatibility Changes in MongoDB 4.0*. Accessed: May 2020. [Online]. Available: https://docs.mongodb.com/manual/release-notes/4.0-compatibility/

[17] (2019). *Prisma–Building the Data Layer for Modern Applications*. Accessed: May 2020. [Online]. Available: https://www.prisma.io/

[18] B. Liskov, "The power of abstraction," in *Turing Award Lecture*. 2009. [Online]. Available: https://amturing.acm.org/vp/liskov_1108679.cfm

[19] (2020). *Redis*. Accessed: May 2020. [Online]. Available: https://redis.io/

[20] (2016). *Apache Cassandra*. Accessed: May 2020. [Online]. Available: http://cassandra.apache.org/

[21] (2020). *C++ Standard Library Headers*. Accessed: May 2020. [Online]. Available: https://en.cppreference.com/w/cpp/header

[22] 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 Conf. Special Interest Group Data Commun.*, London, U.K.:, Aug. 2015, pp. 227–240, doi: 10.1145/2785956.2787501.

[23] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "FASTER: A concurrent key-value store with in-place updates," in *Proc. Int. Conf. Manage. Data*, Houston, TX, USA, May 2018, pp. 275–290, doi: 10.1145/3183713.3196898.

[24] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. 4th Annu. Symp. Cloud Comput. (SOCC)*, Santa Clara, CA, USA, Oct. 2013, pp. 1–15.

[25] D. Lugones, J. A. Aroca, Y. Jin, A. Sala, and V. Hilt, "AidOps: A data-driven provisioning of high-availability services in cloud," in *Proc. Symp. Cloud Comput.*, Santa Clara, CA, USA, Sep. 2017, pp. 466–478, doi: 10.1145/3127479.3129250.

[26] (2019). *Seastar*. Accessed: May 2020. [Online]. Available: http://seastar.io/

[27] (2020). *Data Plane Development Kit*. Accessed: May 2020. [Online]. Available: https://www.dpdk.org/

[28] (2017). *Libevent–An Event Notification Library*. Accessed: May 2020. [Online]. Available: https://libevent.org/

[29] (2017). *Hiredis-Vip*. Accessed: May 2020. [Online]. Available: https://github.com/vipshop/hiredis-vip

[30] (2020). *Datastax C/C++ Driver for Apache Cassandra*. Accessed: May 2020. [Online]. Available: https://github.com/datastax/cpp-driver

[31] (2016). *Dell Poweredge C6320*. Accessed: May 2020. [Online]. Available: https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-PowerEdge-C6320-Spec-Sheet.pdf

[32] (2019). *The Pktgen Application*. Accessed: May 2020. [Online]. Available: https://pktgen-dpdk.readthedocs.io/en/latest/

[33] *IEEE Standard for Ethernet*, IEEE Standard 802.3-2018 (Revision IEEE Standard 802.3-2015), 2018, pp. 1–5600.

[34] (2019). *Intel 82599 10 GbE controller Datasheet*. Accessed: May 2020. [Online]. Available: https://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html

[35] (2016). *Data Plane Development Kit: Performance Optimization Guidelines*. Accessed: May 2020. [Online]. Available: https://software.intel.com/en-us/articles/dpdk-performance-optimization-guidelines-white-paper

[36] (2020). *Testpmd Application User Guide*. Accessed: May 2020. [Online]. Available: https://doc.dpdk.org/guides/testpmd_app_ug/

[37] M. Manesh, K. Argyraki, M. Dobrescu, N. Egi, K. Fall, G. Iannaccone, E. Kohler, and S. Ratnasamy, "Evaluating the suitability of server network cards for software routers," in *Proc. Workshop Program. Routers Extensible Services Tomorrow (PRESTO)*, New York, NY, USA, 2010, pp. 1–6.

[38] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, J. R. Lorch and M. Yu, Eds. Boston, MA, USA: USENIX Association, Feb. 2019, pp. 79–94. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/didona

[39] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker, "A high performance packet core for next generation cellular networks," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Los Angeles, CA, USA, Aug. 2017, pp. 348–361, doi: 10.1145/3098822.3098848.

[40] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. J. Clark, "Kinetic: Verifiable dynamic network control," in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Oakland, CA, USA, May 2015, pp. 59–72. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kim

[41] M. Peuster and H. Karl, "E-state: Distributed state management in elastic network function deployments," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Seoul, South Korea, Jun. 2016, pp. 6–10.

[42] (2019). *nanoSQL—Universal Database Layer for the Client, Server & Mobile Devices*. Accessed: May 2020. [Online]. Available: https://nanosql.io/

[43] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, D. Siewiorek, and M. Satyanarayanan, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, San Jose, CA, USA, Oct. 2017, p. 14, doi: 10.1145/3132211.3134458.

[44] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "Statelet-based efficient and seamless NFV state transfer," *IEEE Trans. Netw. Service Manage.*, vol. 14, no. 4, pp. 964–977, Dec. 2017, doi: 10.1109/TNSM.2017.2760107.

[45] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, doi: 10.1145/1355734.1355746.

[46] (2015). *Apache Java Data Objects*. Accessed: May 2020. [Online]. Available: http://db.apache.org/jdo/

[47] (2020). *Doctrine Database Abstraction Layer*. Accessed: May 2020. [Online]. Available: https://www.doctrine-project.org/projects/dbal.html

[48] (2019). *Storehaus is a Library That Makes it Easy to Work With Asynchronous Key Value Stores*. Accessed: May 2020. [Online]. Available: https://github.com/twitter/storehaus

[49] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire, "Metron: NFV service chains at the true speed of the underlying hardware," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Renton, WA, USA, Apr. 2018, pp. 171–186. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/katsikas

[50] C. Hewitt, P. B. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proc. 3rd Int. Joint Conf. Artif. Intell.* Standford, CA, USA, Aug. 1973, pp. 235–245. [Online]. Available: http://ijcai.org/Proceedings/73/Papers/027B.pdf

[51] J. Duan, X. Yi, S. Zhao, C. Wu, H. Cui, and F. Le, "NFVactor: A resilient NFV system using the distributed actor model," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 586–599, Mar. 2019.

[52] F. Chen, C. Wu, X. Hong, and B. Wang, "Easy path programming: Elevate abstraction level for network functions," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 189–202, Feb. 2018.

[53] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful network-wide abstractions for packet processing," in *Proc. SIGCOMM*, Florianopolis, Brazil, Aug. 2016, pp. 29–43.

[54] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. ATC*, Boston, MA, USA, Jul. 2018, pp. 133–146.

[55] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *Proc. CIDR*, Asilomar, CA, USA, Jan. 2019. [Online]. Available: http://cidrdb.org/

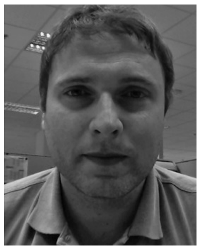
[56] (2020). *Netflix Conductor*. Accessed: May 2020. [Online]. Available: https://netflix.github.io/conductor/

MATTEO POZZA received the bachelor's and master's degrees in computer science from the University of Padua, Italy, in 2014 and 2016, respectively, and the Ph.D. degree from the University of Helsinki, in 2020.

His research interests include theoretical and practical problems in networked systems, especially mobile networks.

ASHWIN RAO received the Ph.D. degree from the DIANA (formerly, Planete) Project Team, Inria Sophia Antipolis, and the master's degree from the School of Information Technology, IIT Delhi.

He is currently a Docent (Adjunct Professor) with the University of Helsinki. His research interests include communication networks, distributed systems, privacy, building next generation distributed systems, and performing measurements for getting insights on the dynamics of communication networks with the objective of identifying performance and privacy issues.



DIEGO F. LUGONES (Member, IEEE) received the B.S. degree in engineering from La Plata National University, Buenos Aires, Argentina, and the M.Sc. and Ph.D. degrees in high performance computing from the Autonomous University of Barcelona, Barcelona, Spain.

From 2006 to 2009, he was a Professor of computer science with the Autonomous University of Barcelona. He worked with the Exascale Computing Laboratory, HP Labs, Barcelona, in topics related to simulation frameworks for communication analysis. He is currently the Head of the application platforms and software systems research with Nokia Bell Labs, Dublin, Ireland, where he leads the research on software systems and technology prototyping of emerging cloud infrastructures with a focus on artificial intelligence (AI) enabled management for predictive and multi-variable orchestration. Before joining Nokia Bell Labs, he worked with IBM Exascale Research, Dublin, on topics related to data center interconnects, specifically on hybrid optical/electronic-circuit/packet-switching network architectures.

SASU TARKOMA (Senior Member, IEEE) is currently a Professor of computer science with the University of Helsinki and the Head of the Department of Computer Science. He has authored four textbooks and has published over 200 scientific articles. He holds nine granted U.S. patents. His research interests include Internet technology, distributed systems, data analytics, and mobile and ubiquitous computing. He is a Fellow of the IET and EAI. His research has received several best paper awards and mentions in conferences and publications, such as the IEEE PerCom, the IEEE ICDCS, ACM CCR, and ACM OSR.