E-State: Distributed State Management in Elastic Network Function Deployments

Manuel Peuster
University of Paderborn
manuel.peuster@uni-paderborn.de

Holger Karl
University of Paderborn
holger.karl@uni-paderborn.de

***Abstract*—Elastic deployments of virtualized network functions (VNFs) can automatically scale the amount of used resources in relation to their workload. This is often done by starting new VNF instances or stopping old ones. A problem of these scale operations is that most network functions are stateful and their internal state is not automatically migrated when traffic is redistributed. As a result, mechanisms are needed to exchange or migrate internal network function state between VNF instances. This paper presents a state management framework that creates logically distributed state memory on top of elastically deployed VNFs used to share state information between these VNFs. We introduce a novel programming model that provides both a local and a global view of the state to each VNF instance. Further, we compare the performance of our prototype to a centralized and a distributed in-memory database solution.**

I. Introduction

Network function virtualization (NFV) implements network functions, previously deployed as dedicated hardware boxes, in software to execute them in virtual machines on cloud infrastructure providing the benefit that additional resources can be added or removed on-demand [1], [2]. This process can be fully automated and the number of virtual instances of a particular virtualized network function (VNF) can be changed in relation to its workload, e.g., amount of processed traffic.

One of the main problems in creating such elastic VNF deployments is the fact that many network functions are stateful. Typical examples for this are network address translation boxes (NAT) that store mappings between ports and hosts or intrusion detection systems (IDS) that keep track of pattern matchings to detect attacks. The typical structure of this network function application state can be divided into two classes [3]. The first class contains *global state* accessed independently of the processed traffic. The second class contains *partitioned state* that consists of chunks of state directly related to one or multiple network flows or sessions processed by the network function. These state chunks can be identified with the same information used to identify single flows or sessions. For IP connections this is done by 5-tuples consisting of source IP, target IP, source port, target port, and protocol. In typical network functions most parts of the application state is represented by the second class which can easily be distributed across multiple VNF instances [3], [4].

A problem appears when an elastic VNF deployment is changed and VNF instances are dynamically added to (*scale-out*) or removed from (*scale-in*) the system. In such cases, the flow or session assignments are changed to rebalance the traffic in the system which leads lost state information on some VNF instances.

One obvious solution to handle this in the scale-out case is assigning only new connections to recently added instances and keep existing connections on old instances that already contain the corresponding state. Even though this solutions is easy to implement, it may lead to imbalanced load situations because connections are not moved away from overloaded instances. For the scale-in case, the obvious solution is to keep instances in the system as long as there are ongoing connections assigned to them. But this comes with the downside that scale-in operations may be blocked for an unpredictable amount of time by long living connections. These problems create the need of a state management system which is able to automatically share and move application state between VNF instances.

In this paper, we introduce the *E-State framework*. This framework provides a novel approach to share application state between elastically deployed VNF instances by using logically distributed state memory that is accessed by each VNF instance. With this solution, no central control application is needed and the system becomes more fault-tolerant and scalable. We introduce a proof-of-concept implementation of our system and compare it to three other approaches: A system without state management, a system with centralized state memory, and a system which uses a generic distributed memory solution not optimized for VNF state management.

The rest of this paper is organized as follows. We present existing solutions in Section II. Section III explains our system design and Section IV describes our prototype implementation and its comparison to other solutions. Section V concludes.

II. Related Work

This section presents three state management solutions that explicitly exploit the structure of network function state before discussing similarities and differences to our solution.

In the first approach, Olteanu et al. [5] propose a mechanism that is inspired by virtual machine migrations. It moves network function state from one VNF instance to another in three steps. In the first step, the complete state is copied to the target instance so that this instance is ready to process new flows. In the second step, all new flows are forwarded from the source instance to the target instance which processes them and allocates new state. The third step freezes the processing of the remaining old flows on the source instance and moves their flow-related state to the target instance. At the end, old flows are redirected to the target instance by changing forwarding rules on an SDN switch.

The second solution is called Split/Merge [3]. It is implemented as a shared library that acts as a memory allocator for network functions. It exposes an API that allows allocating flow-related and globally shared state. A central controller decides which state should be moved between the instances. The approach implements a mechanism to merge state by using custom combiner functions defined by the network function developer.

The third framework is called OpenNF [6]. It provides coordinated control of network function state and network forwarding rules. This framework uses a central management application to move state and flows from one instance to another. To integrate a VNF into the system, it has to implement a set of API functions to pull and push state information. When the central management application decides to move a flow from one instance to another, it pulls the state from the source instance and pushes it to the target instance. During this process, arriving packets are buffered at the controller until the state is transferred to the target. The buffered packets are then forwarded to the target instance. By using these mechanisms, OpenNF is able to perform loss-free and order-preserving flow moves. Two extensions of this approach [7], [8] add solutions for direct state transfers between VNFs to protect the controller from becoming the bottleneck when incoming packets are buffered. However, the system management remains centralized in both extensions.

All these approaches utilize a centralized control component to decide which parts of state are moved between VNF instances. Our framework is unique in not relying on such a centralized state management controller; rather, it utilizes logically distributed state memory to receive state information from other instances when they are needed. Olteanu et al. [5] do provide a simple migration mechanism for VNF state. They do not provide solutions to share global state between VNF instances and each state item is always only visible on exactly one instance. Split/Merge [3] provides custom combiner functions to merge state from different instances which is also possible with our solution. But Split/Merge's combiner functions are only executed when flows are consolidated on one VNF instance and are not used to provide a global view on the entire state space when needed. OpenNF [6] uses central applications to control the state management. This requires knowledge about the VNFs running in the system to decide which parts of the state should be moved. In contrast, our system reacts to flows moving in the underlying network and does not depend on state management decisions taken outside of the VNF instances.

Other more generic solutions to share common information between VNF instances are distributed memory systems, like a REDIS Cluster [9] or Apache Cassandra [10]. These approaches provide good scaleability but have no notion about the structure of the managed state. Our solution, in contrast, explicitly exploits the state structure and keeps flow related information on the VNF instance that needs it.

III. A Distributed State Management Framework

We introduce the *E-State framework*, a flexible and scaleable state management solution that enables elasticity for stateful VNFs. E-State is built as a software library used to access and share state information. In our design, every VNF instance becomes one node of the distributed state memory and thus the system automatically scales with the number of VNF instances.

A. State Management with Global View

A network function can use *E-State* to store arbitrary chunks of state data, e.g., a serialized data structure representing a runtime object. We call these chunks *state items*. A simple solution to share state items between VNF instances would be to allocate them in a distributed data structure and write all updates directly to this structure. The obvious problem of this approach is the additional delay that is introduced when one VNF instance frequently reads or writes items stored on another instance[1].

A better solution is to exploit the fact that most state items are directly related to processed flows or sessions. All accesses to these state items are performed by a single VNF instance and accesses to other VNF instances are only needed when the traffic assignment changes. As a result, *E-State* provides an access pattern offering fast reads and writes to flow-related state items of the local VNF instance and the possibility to read state items on other instances when needed.

1) System Design: In the E-State framework each VNF instance stores all its state items in its own *local state memory* which is never written by other VNF instances. This results in small access delays and ensures that each VNF instance has a *strictly consistent* view of its own state items.

Even though this simple design would ensure that the internal network function state is visible to our system, it does not yet provide a solution to exchange state items between VNF instances. To overcome this, the system offers a special read operation that allows a network function to request state items from all VNF instances in the elastic deployment. Using this, a network function is always able to request *global state* information about the entire elastic system. An example for this, is receiving the match counter value from each VNF instance of an elastic IDS system. The only thing the network function developer has to take into account when global reads are used is that they provide an *eventual consistency* model instead of the strict consistency model provided by local operations. However, our framework offers the flexibility to implement stricter consistency models so that it can be used as an experiment platform for different state sharing approaches.

[1]Similar to page trashing in distributed shared memory.

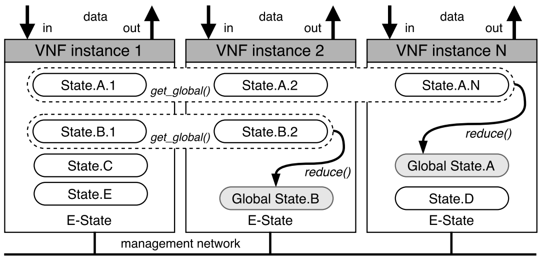

Fig. 1: State management with local and global view

2) Reduce Operation: A global read operation receives a list that contains up to n state items where n is the number of VNF instances in an elastic deployment. Such a list does not provide a consolidated view on the system and needs some processing to compute a *global view* of the requested state items. To do so, network function developers can specify a *reduce* function that maps a list of state items to a single, consolidated state item representation. A typical example for this is a reduce function that is passed a list of counter values and returns their sum or average. This concept is comparable to custom merge functions presented in [3] but provides more flexibility since a global read can be applied whenever a network function needs it.

Figure 1 demonstrates the *E-State* concept. It shows three instances of a network function (e.g. three VM instances or containers) all linking against the *E-State* library connected by a management network. Each instance allocates different state items in its local state memory depending on the flows they process (*State.A* - *State.E*). Some state items appear on multiple instances since a local version of the contained state is allocated by each VNF instance. The figure shows how these replicated state items can be fetched by other instances with the *get_global* operation and how a consolidate state item that reflects the global view on the system is computed on-the-fly by *reduce* functions.

Such reduce functions can not only be used to combine state items but also to select a particular item out of the collection of state items stored on different instances. The most common use case for this is finding the state item that was updated most recently. To do so, a reduce function needs a happen-before relationship between state item updates which can, e.g., be based on real-time timestamps with the risk to produce wrong relationships caused by clock drifts. A better solution for this is using a vector clock mechanism which provides happen-before relationships between state items on different instances [11] with the downside of additional communication overhead for synchronization.

3) Flow Reassignments: The main use case of our state management system are scenarios in which VNF instances are added or removed and the flow assignment is changed. In such cases, redirected flows appear on their target VNF instance, which needs to fetch the corresponding flow-related state items from the source instance. This is done with the global read operation and does not require an explicit fetch or move functionality used by other approaches [3], [6]. The SDN controller can optionally support this process by marking packets of moved flows, e.g., by setting their VLAN tags. Using this, the target VNF instance can easily distinguishing new flows from redirected flows and does only need to perform global reads when a redirected flow is detected.

B. Programming model and API

The *E-State* API is inspired by a key-value store and provides three basic functions: *set*, *get*, and *delete*. Further, it provides a *get_global* function. They are defined as follows:

- **`set(key, state_item)`** Creates or updates state items stored locally in the shared library.
- **`get(key):state_item`** Returns the value of a state item stored in the local library. This gives a local view to the system.
- **`del(key)`** Removes the specified state item from the local state store.
- **`get_global(key, red_func):state_item`** Returns the result of the specified reduce function that is applied to all state items stored on all connected VNF instances matching the given key. This function has no side effects on any VNF instance and does not change state items.

The global view is requested with the `get_global` API call that is passed a reduce function pointer as second parameter. Such a reduce function has to have the following signature:

- **`red_func(list<state_item>):state_item`**

The function is passed a list of state items and returns a combined representation of them. This allows network function developers to specify how the mapping from multiple state items to a consolidated global view should be done. It is recommended that the given reduce function is commutative since the order of passed state items is not fixed and may change between calls.

Our system uses arbitrary strings as keys to identify different state items. We do not fix the used key structure and leave it to the network function developers to specify their own schemes (e.g., 5-tuples to identify flows). These keys are checked for equality when a specific state item is requested. In addition to this, the `get_global` function allows to use wildcard symbols in its keys based on regular expressions that are matched against existing keys.

IV. Prototype Implementation

This section describes the prototype implementation of our *E-State framework* and compares it to a system without state management as well as to a centralized and a distributed state memory solution [12].

A. System Design

The main component of our system is a shared library, called *libestate*, that is implemented in C++ and offers a standard C interface against which network function applications can link. The interface offers all functions described earlier, including a `get_global` function that expects a pointer to

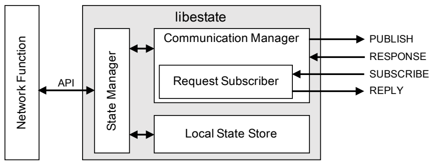

Fig. 2: Design of our shared library including a communication manager that interacts with other libestate instances

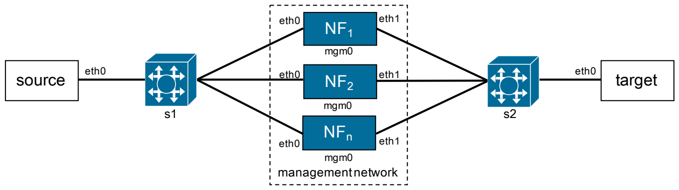

Fig. 3: Mininet topology used for prototype evaluation

a custom reduce function or the name of a predefined reduce function as one of its arguments.

Figure 2 shows the main modules of our library. The *State Manager* is responsible for providing the interface to the network function and to control all internal procedures. It interfaces with the *Local State Store*, which is a key-value store responsible for holding state items registered by a network function. To enable our library to receive state items form other instances and thus to obtain a global view of the entire state space, we introduce a third module called *Communication Manager*. This module uses the distributed messaging system ZeroMQ [13] as communication backend. It contains a *Request Subscriber* module which runs in an independent thread and replies to state requests from other instances.

E-State uses a publish/subscribe communication pattern together with ZeroMQ's push/pull pattern to do global state requests. To do so, each network function instance is always subscribed to all other instances of the same elastic deployment. Each instance is then able to publish `get_global` requests and the other instances reply to it.

B. Prototype Evaluation

We tested our *libestate* prototype in a Mininet [14] environment running on a machine with Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz and 16GB memory. Figure 3 shows the topology used for our experiments. It consists of a source host that sends iperf-generated TCP traffic to a target host over an elastic cluster of VNF instances which forward and monitor the traffic. These VNF instances are Mininet hosts with two ethernet interfaces configured as ethernet bridges. All VNF hosts are also connected to a management network for communication between *libestate* instances. The two SDN switches between source and target are controlled by a custom SDN application running on top of a POX controller which proactively installs forwarding rules on the two switches to control the traffic distribution and flow moves between available VNF instances.

Each network link in our topology has set a maximum bandwidth of 1 Gbit/s and no artificial delay. We use Mininet's CPU sharing limitation feature for each Mininet host to emulate a realistic scenario in which an additional VNF instance corresponds to additional computation resources. Without this, a higher number of VNF hosts in the system would not result in performance improvements because the CPU time available for each single Mininet host would decrease. We limited the hosts as follows: Source and target host are limited to 20% CPU each and every VNF instance is assigned to 2.5% of the overall CPU time summing up to 40% CPU usage when the maximum of 16 VNF hosts are active. The remanning 20% are used for other components, like the SDN controller and the centralized state memory used in one of our experiments. We use a custom network function implementation that runs in each VNF host and performs pattern matchings on sniffed traffic like an IDS.

Our first experiment demonstrates how an IDS can benefit from our state management system when it is scaled out and in at runtime. For the scale-out case, the experiment starts with a single VNF instance (*NF.1*) over which all TCP flows are forwarded. After about 55 seconds, the scale procedure is initiated and half of the flows are rerouted to a new VNF instance (*NF.2*) by installing additional rules on the two SDN switches. Figure 4 (top) shows the scale-out scenario. The vertical dashed line marks the point in time at which scaling starts. The left part of the figure shows the values of the pattern match counter on both instances. The experiment is executed two times. At first, with a *baseline* system without any state sharing functionality and second with our *libestate* system. In the *baseline* case, the second instance starts its match counters from zero after flows are moved to it, even though the first instance has already detected intrusive packets. This might lead to missed detections and influences the correctness of the overall IDS system. In the *libestate* case, the state for the moved flows is transferred to the second instance and the operation can continue without information loss. The right part of the figure shows how the overall performance of the elastic VNF deployment increases after the system is scaled out. Figure 4 (bottom) shows the scale-in case in which the experiment starts with two VNF instances and then moves all flows to *NF.1* after about 55 seconds. It shows how the match counter of *NF.2* stops counting in the *baseline* version and the information is lost.

The second experiment evaluates the scaling behavior of our state management system with an elastic deployment of 2 to 16 replicated VNF instances. It compares our *libestate* prototype to three other approaches using the average number of processed packets per second to show the overall system performance and the average state item request delay to show the state sharing performance. First, we compare to the default IDS implementation not using any state management mechanisms, called *baseline*[2]. Second, to a state management system (*centralmem*) that uses a single REDIS instance [15] to

[2]Global values of baseline experiment are calculated offline by summing up the local values logged on each VNF instance.

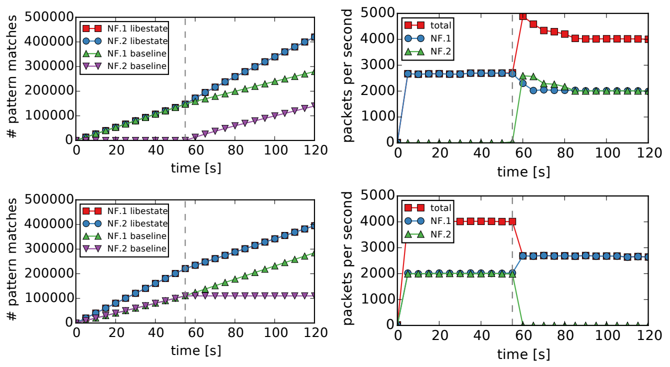

Fig. 4: Match counter value of two IDS instances (left) and overall system performance before and after scale operation (right). Scale-out case (top) and scale-in case (bottom).

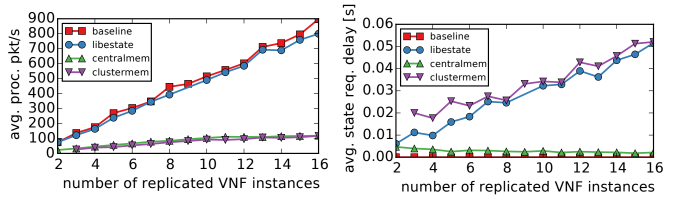

Fig. 5: System performance as number of processed packets per second (left) and state item request delay (right) for different numbers of replicated VNF instances

maintain the state of all VNF instances. Third, we compare to a distributed REDIS cluster (*clustermem*) that runs one REDIS node on each VNF instance and can be used through the same API like the centralized version.

The left plot of Figure 5 shows how the performance of the IDS system increases when additional instances are added and how the scaleability of a system with centralized memory is limited. This is, on the one hand, caused by additional delay introduced by turning each state access into a network request, and on the other hand, by the maximum number of requests a centralized solution can serve. It also shows that a distributed memory solution does not provide benefits because state items are not stored on the VNF instances which access them most often. The performance of our library, in contrast, is near to the baseline performance since most of the state accesses are done locally and the global state is requested less often. It is important to note that the baseline implementation does not share any state information when flows are moved. Our system, in contrast, maintains all state information and provides comparable performance, which is a clear advantage. However, an increasing number of VNF instances results in an increased delay for each global request performed by our library (Figure 5, right). This is expected because the system always requests state items from all instances of an elastic VNF. It is also interesting that the request delays of the *clustermem* version are higher than the delays of the *centralmem* version. The reason for this is that state items have to be fetched from multiple cluster instances to obtain the global view instead of requesting all items at once from the centralized memory.

V. Conclusion

This paper presented *E-State*, a novel approach to manage application state in elastic network function deployments. Our solution shows that NFV state management can be done without central control components which are used by existing approaches [3], [6]. The presented prototype is a first step towards a generic, distributed state management framework that does not rely on centralized control mechanisms. Our prototype can easily be extended to provide additional consistency models for state requests to study tradeoffs between consistency requirements and management overhead. The results of our experiments show that our system outperforms approaches that use generic centralized or distributed state memory solutions.

Acknowledgments

This work has been partially supported by the SONATA project, funded by the European Commission under Grant number 671517 through the Horizon 2020 and 5G-PPP programs (www.sonata-nfv.eu) and the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901).

References

[1] ETSI, “NFV White Paper,” https://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.
[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” *Communications Magazine, IEEE*, vol. 53, no. 2, pp. 90–97, 2015.
[3] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, “Split/Merge: System Support for Elastic Execution in Virtual Middleboxes.” in *NSDI*, 2013, pp. 227–240.
[4] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, “Toward software-defined middlebox networking,” in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 7–12.
[5] V. A. Olteanu and C. Raiciu, “Efficiently migrating stateful middleboxes,” in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 93–94.
[6] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, “OpenNF: Enabling innovation in network function control,” in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 163–174.
[7] A. Gember-Jacobson and A. Akella, “Improving the safety, scalability, and efficiency of network function state transfers,” in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, 2015, pp. 43–48.
[8] B. Kothandaraman, M. Du, and P. Sköldström, “Centrally controlled distributed vnf state management,” in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, 2015, pp. 37–42.
[9] The Redis Project, “Redis key-value cache and store,” http://redis.io, 2015.
[10] The Apache Software Foundation, “Cassandra Distributed Database,” http://cassandra.apache.org, 2015.
[11] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[12] M. Peuster, “E-State Prototype,” https://github.com/mpeuster/estate, 2015.
[13] iMatix, “ZeroMQ Distributed Messaging,” http://zeromq.org, 2015.
[14] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.
[15] Redislabs, “Redis in-memory store,” http://redis.io, 2015.