Realizing Highly-Available, Scalable and Protocol-Independent vSDN Slicing with a Distributed Network Hypervisor System

Huibai Huang, Bin Niu, Shaofei Tang, Shengru Li, Sicheng Zhao, Kai Han,
Zuqing Zhu, *Senior Member, IEEE*

***Abstract*—In this paper, we design and implement a distributed network virtualization hypervisor (NVH) system, namely DPVisor, which can provide superior network programmability based on protocol-oblivious forwarding (POF) to realize highly-available, scalable and protocol-independent virtual software-defined network (vSDN) slicing. The experimental comparisons indicate that DPVisor achieves comparable performance as ONVisor (*i.e.*, an OpenFlow-based ONOS benchmark), in terms of the message processing latency, message processing throughput, and failure recovery time. Moreover, to optimize the performance of distributed NVH systems, we carefully adjust the consistency model used in the state synchronization of NVH instances and propose a local cache based scheme to balance the tradeoff between the consistency and availability of network status. Our experimental results confirm that the message processing throughput of distributed NVH systems can be greatly improved (*i.e.*, for both DPVisor and ONVisor), while the data consistency among NVH instances is still maintained well.**

***Index Terms*—Software-defined networking (SDN), Network virtualization, Distributed network hypervisor, Consistency model, Protocol-oblivious forwarding (POF).**

I. Introduction

RECENTLY, network virtualization has attracted intensive interests from both academia and industry [1, 2], since it allows an infrastructure provider (InP) to dynamically slice logically-isolated virtual networks over a shared substrate network according to the demands from service providers (SPs) [3, 4]. Meanwhile, SPs can operate the virtual networks (*i.e.*, the tenants) by leveraging the principle of software-defined networking (SDN) to ensure flexibility and network programmability. Therefore, by combining network virtualization with SDN, the InP can slice virtual SDNs (vSDNs) [5] and offer SPs the flexibility of running their own protocols and applications on the vSDNs [6–9], to accelerate the deployment of new services and facilitate network innovations.

Although network virtualization with vSDNs enhances the network system's performance in terms of programmability and adaptivity, the task can never be accomplished without an effective network virtualization hypervisor (NVH) system [10]. Specifically, the NVH system needs to abstract the resources in the substrate network, virtualize substrate switches for the tenants, and bridge the communications between the substrate switches and the vSDNs' controllers. Hence, from the perspective of each controller, it directly manages the virtual switches in its vSDN, while all the substrate switches view the NVH as the centralized controller for flow management. Previously, people have developed a few NVH systems to realize vSDN slicing, *e.g.*, FlowVisor [11], OpenVirteX [6], and SR-PVX [8]. Even though these NVH systems have been proven to be effective, there are still a few unaddressed issues that restrict their practicalness, especially for being used in a large-scale, complicated and heavy-loaded substrate network.

H. Huang, B. Niu, S. Tang, S. Li, S. Zhao, K. Han, and Z. Zhu are with the School of Information Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, P. R. China (email: zqzhu@ieee.org).

Manuscript received August 26, 2017.

This is because the NVH systems in [6, 8, 11] were all based on a centralized architecture, which means that a single NVH sits in between the vSDNs' controllers and substrate switches to handle all the flow management messages. This, however, makes the system vulnerable to NVH failures, *i.e.*, a single failure on the NVH can bring down the services of all the vSDNs. Therefore, the centralized architecture cannot guarantee high-availability, which is known to be vital in network design, especially for wide-area networks [12, 13]. Moreover, since it needs to translate and forward all the flow management messages, the single NVH can become the performance bottleneck of the network system. In order to address these reliability and scalability issues, one can try to design a logically centralized but physically distributed architecture for the NVH system, similar to the efforts made for the control plane of SDN [14, 15]. Nevertheless, the design and implementation of such a distributed NVH system still have not been fully explored before.

In this work, we design and implement a distributed NVH system to realize highly-available, scalable and protocol-independent vSDN slicing. Specifically, the NVH system takes the architecture in Fig. 1 and is developed based on the well-known ONOS platform [15]. By following our previous idea in [8], we first expand the NVH module in ONOS to add in the support of protocol-oblivious forwarding (POF) [16]. Hence, the programmability and flexibility of the NVH system can be significantly improved to accomplish protocol-independent vSDN slicing. Then, we transform the centralized NVH system into a distributed one by creating distributed NVH instances and letting them use east/west-bound communications to maintain a logically centralized view of the substrate network and vSDNs. In other words, each NVH instance only needs to manage a subset of the substrate network directly, and all the instances communicate with each other to synchronize network status. Therefore, the workload can be distributed to the NVH instances to achieve better scalability, and when an NVH instance breaks down, another one can take over its management tasks instantly for realizing high-availability.

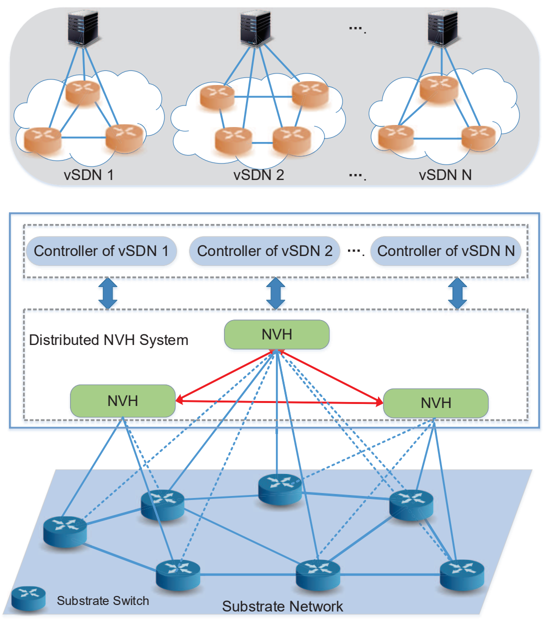

Fig. 1. Architecture of distributed NVH system for vSDN slicing.

Next, we conduct experiments to compare our distributed NVH system, namely DPVisor, with the original OpenFlow-based ONOS benchmark (*i.e.*, ONVisor [17, 18]), in terms of the message processing latency, message processing throughput, and failure recovery time. The results demonstrate that DPVisor achieves comparable performance as ONVisor for all the three metrics. This confirms that our design adds in POF support to significantly improve the programmability and flexibility of the distributed NVH system without causing noticeable performance degradation. Finally, to optimize the performance of the distributed NVH system, we carefully adjust the consistency model used in the state synchronization of NVH instances. Specifically, we propose a local cache based scheme to balance the tradeoff between the consistency and availability of network status. As a result, the message processing throughput of distributed NVH system can be greatly improved (*i.e.*, for both DPVisor and ONVisor), while the data consistency among NVH instances is still maintained well. Our major contributions can be summarized as follows.

- We design and implement DPVisor, which is the first distributed NVH system that can realize highly-available, scalable and protocol-independent vSDN slicing.
- We experimentally demonstrate the scalability and availability of DPVisor with a real network testbed. To the best of our knowledge, this is the first demonstration of such a distributed NVH system. Experimental results verify that DPVisor can achieve comparable performance as its OpenFlow-based counterpart (*i.e.*, ONVisor).
- We carefully analyze the performance bottleneck of the distributed NVH systems based on ONOS (*i.e.*, both DPVisor and ONVisor), and design a local cache based scheme that can not only improve the systems' message processing throughput significantly but also maintain the data consistency among NVH instances well.

The rest of the paper is organized as follows. Section II provides a brief survey of the related work to explain the background and motivation of this work. The design and implementation of our distributed NVH system are described in Section III. We discuss the experimental evaluations of our proposed NVH system in Section IV, and the performance optimization and related results are presented in Section V. Finally, Section VI summarizes the paper.

II. Related Work

A. Network Virtualization Hypervisors (NVHs)

The NVH system is a key component to realize vSDN slicing, and for a comprehensive survey on the previous studies on NVH, one is recommended to refer to [10]. FlowVisor [11] was the first NVH system that can create OpenFlow-based vSDNs over a shared substrate network. To isolate the message processing in different vSDNs, FlowVisor defines the concept of "flowspace" to represent the subsets of OpenFlow-supported header fields. This means that the vSDNs created by FlowVisor cannot use overlapped header space, which greatly limits the scalability and flexibility of vSDN slicing. Another drawback of FlowVisor is that it can only create vSDNs whose topologies are the same as that of the substrate network. Al-Shabibi *et al.* [6] successfully addressed the limitations of FlowVisor by designing and implementing OpenVirteX, which inherits the overall architectural design of FlowVisor but resolves the issues caused by the flowspace. Specifically, OpenVirteX allows the tenants to use overlapped header space and specify the topologies for their vSDNs.

However, FlowVisor and OpenVirteX are both based a centralized architecture, which means that the InP would only use a single NVH instance to manage all the vSDNs. To address the reliability and scalability issues caused by the centralized architecture, a distributed NVH system is desired [19]. In [20], Bozakov *et al.* proposed AutoSlice, which uses multiple FlowVisor instances to act as the NVH system and distribute the workload in multiple SDN domains. However, since AutoSlice relies on a single management module to conduct vSDN slicing in each SDN domain, it is not a truly distributed system and thus cannot ensure high availability. In [21], the authors laid out the design of a distributed NVH system, namely, FlowN. However, FlowN only provides a container-based controller for tenants to install the applications for their vSDNs, which means that each tenant cannot deploy its own vSDN controller in FlowN. This limits the flexibility of vSDN slicing. Moreover, since FlowN runs all the tenant applications on a single container-based controller, it is actually not a truly distributed NVH system. Lastly but not least, the authors only discussed the architectural design of FlowN in [21] but did not include any experimental results.

Note that, FlowVisor, OpenVirteX and FlowN are all based on the OpenFlow protocol [22], which is the most popular protocol to support SDN. Nevertheless, as OpenFlow-based match fields are all based on the existing network protocols, an OpenFlow switch can only parse and match to the fields that have already been standardized in the OpenFlow specifications. This would make the data plane protocol-dependent and cause compatibility issues. The same issues limit the programmability and flexibility of OpenFlow-based NVH systems, *i.e.*, it would be difficult to create vSDNs to support new protocols that have not been included in the OpenFlow specifications. For instance, even though the OpenFlow specifications have already evolved to version 1.5 that includes 44 match fields, FlowVisor, OpenVirteX and FlowN were all developed based on OpenFlow v1.0, which only includes 12 match fields.

The protocol-dependent and compatibility issues induced by OpenFlow can be resolved by leveraging the efforts made to realize protocol-independent forwarding (PIF) [23]. More specifically, by incorporating protocol-oblivious forwarding (POF) [16] in the NVH system, we realized SR-PVX in [8], which allows tenants to customize the packet forwarding behaviors in their vSDNs in an arbitrary manner. The basic idea of POF is straightforward, *i.e.*, it first abstracts all the packet fields to be processed in the data plane as {*offset*, *length*} tuples, where *offset* tells the start bit-location of a packet field and *length* indicates the field's length in bits, and then defines a flow instruction set (*i.e.*, POF-FIS [16, 24]) to assist the packet processing in POF switches. As all the instructions in POF-FIS locate data in packets with {*offset*, *length*} tuples, the POF switches can operate on any bits in packets, without being restricted by pre-defined protocols. Hence, POF can greatly enhance the data plane programmability for SDN, which has already been experimentally demonstrated in our previous studies [24–30]. However, as SR-PVX was developed based on OpenVirteX, it also takes the centralized architecture and cannot avoid the resulting reliability and scalability issues.

B. Distributed Control Plane Systems

Recently, ONOS [15] has been developed as an open-source network operating system that is based on a distributed architecture motivated by the performance, scalability and availability requirements of large-scale networks. A recent project in ONOS has tried to expand its scope on network virtualization and realized a distributed NVH system (*i.e.*, ONVisor) for ONOS [17]. Nevertheless, ONVisor was still developed based on OpenFlow, and could not realize protocol-independent vSDN slicing. More importantly, it is known that ONVisor uses the strong consistency model in the s-tate synchronization of NVH instances, which might affect its processing throughput as we will explain in this paper. Hence, we should also evaluate alternative ways for the state synchronization of NVH instances to see whether a better tradeoff between data consistency and processing throughput would be feasible for distributed NVH systems.

Note that, in a distributed system, the consistency model of data synchronization can take different forms to balance the tradeoff between the consistency and availability of data [31]. Among them, the strong consistency model, the eventual consistency model, or a hybrid combination of them have been

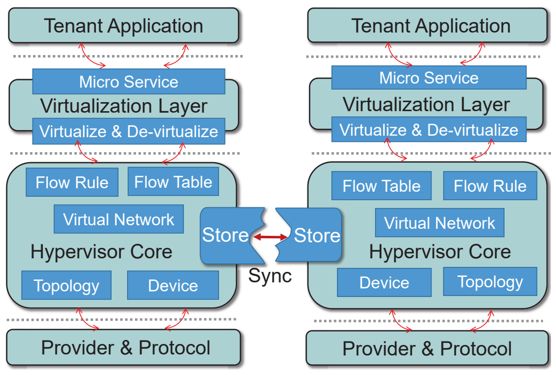

Fig. 2. Architectural design of DPVisor.

frequently considered in realizing distributed control plane systems for SDN. The strong consistency model ensures that each instance always reads the most updated data, and thus if certain data has not been updated to all the instances, it cannot be read. The Raft consensus protocol [32] implemented in ONOS takes the strong consistency model, which would apparently give too much priority to consistency and limit the system's processing throughput due to lack of data availability. Another alternative is the eventual consistency model, which does not enforce that each instance read the most updated data. Instead, the data updates on an instance will be visible to all the others eventually through data synchronization among the instances. Although the eventual consistency model can make data available quickly to achieve relatively high processing throughput, it might sacrifice consistency too much. To the best of our knowledge, how to design a hybrid combination of the consistency models for the distributed NVH system, which can optimize the tradeoff between the consistency and availability of data has not been studied before.

III. System Design and Implementation

A. Distributed NVH System based on ONOS

Fig. 2 shows the architectural design of DPVisor, which is a distributed NVH system designed based on ONOS. Here, each NVH instance consists of four major modules as follows.

- *Provider & Protocol:* This module is in charge of the south-bound protocol stack, and it communicates with substrate switches and forwards the obtained network status to the Hypervisor Core. We modify the original Provider & Protocol module of ONOS to add in the support of POF. Hence, the POF-based flow entries and flow tables can be encoded in *Flow_Mod* and *Table_Mod* messages, respectively, and sent to the substrate POF switches by this module.
- *Hypervisor Core:* This module realizes the abstraction of the substrate network. Specifically, all the substrate network elements (*e.g.*, switches and links) and their statistics are collected by the Hypervisor Core, and they are stored and synchronized in the NVH instances according to certain consistency models. Therefore, the NVH instances can have a logically centralized view of the network. Fig. 2 also shows the submodules in the

Hypervisor Core. Among them, the Device and Topology submodules handle the network status regarding the substrate switches and topology, respectively, the Flow Rule submodule builds the flow entries to be installed in the substrate switches for parsing and processing packets, the Flow Table submodule translates the protocols defined by tenants into POF-based flow tables, and the network virtualization with vSDNs is realized by the Virtual Network submodule.

- *Virtualization Layer:* This module realizes the major functionalities for network virtualization. First of all, the Micro Service submodule allows each vSDN to specify its own service, which is identified by the Virtualization Layer according to the unique tenant ID of the vSDN. Secondly, the packets/messages from the substrate network carry their tenant IDs, based on which they will be forwarded to the corresponding vSDNs' services by the Virtualize & De-virtualize submodule. In the opposite direction, the Virtualize & De-virtualize submodule translates all the messages from the vSDNs' services into what the substrate switches can understand based on the mapping between the virtual and substrate networks.
- *Tenant Application:* This module carries the services of the vSDNs, which works as a controller instance for each vSDN. Since the messages to/from this module are categorized by the Micro Service submodule in the Virtualization Layer, our design ensures that the operations of the vSDNs are independent and isolated. Moreover, as each NVH instance includes a Tenant Application module, the actual implementation of each vSDN controller is also physically distributed but logically centralized. Therefore, our DPVisor is a truly distributed NVH system.

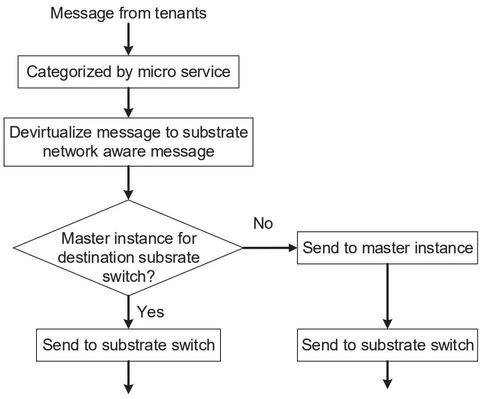
Message from tenants
Categorized by micro service
Devirtualize message to substrate network aware message
Master instance for destination subsrate switch?
No
Send to master instance
Yes
Send to substrate switch
Send to substrate switch

Fig. 3. Message processing procedure in DPVisor.

B. Message Processing in DPVisor

When DPVisor receives a message from the data plane, it will virtualize and dispatch the message to the corresponding vSDN according to the input port. Messages belonging to different vSDNs are categorized by the isolated micro services. On the other hand, the Virtualization Layer accesses the virtual network store in the Hypervisor Core to de-virtualize the messages from the vSDNs' controllers. The de-virtualization includes the translation of the virtual switch IDs and output ports in the messages to substrate switch IDs and output ports, respectively. Moreover, if the virtual output port in a *Flow_Mod* message is connected to a virtual link, DPVisor will add extendable POF instructions in the *Flow_Mod* message for inserting the corresponding tenant ID and link ID in it. Fig. 3 shows the overall message processing procedure in DPVisor, which indicates that once the *Flow_Mod* messages are de-virtualized in the Virtualization Layer, they will be dispatched to the Flow Rule submodule in Hypervisor Core in Fig. 2. Then, the Flow Rule submodule will find the master NVH instances of the messages' substrate switches and send the messages to them if necessary. Finally, the *Flow_Mod* messages will be sent to the substrate switches by the Provider & Protocol modules in the master instances.

C. Packet Processing for Network Virtualization

Fig. 4 illustrates the packet format that we design to realize vSDN slicing with DPVisor. Specifically, we insert a Virtual Network Header field (*i.e.*, 6 bytes) after the Ethernet header of each packet to assist the packet processing for network virtualization. The field consists of two sub-fields, *i.e.*, the 3-byte Tenant ID sub-field to identify the vSDN and the 3-byte Link ID sub-field to determine the packet's output virtual link. Hence, DPVisor can easily categorize the flows in the substrate network according to their Virtual Network Headers. According to the principle of POF, the Virtual Network Header can be represented by a tuple of $\{112$ bits, 48 bits$\}$, and it can be added, modified and deleted by leveraging the POF-FIS. Note that, since we just insert the Virtual Network Header after the Ethernet header but do not overwrite any existing packet field, the network virtualization with DPVisor is transparent to upper-layer protocols. Therefore, protocol-independent vSDN slicing is feasible, which is realized with the procedure in Fig. 5 in each substrate POF switch.

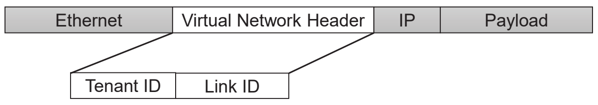

Fig. 4. Packet format to realize vSDN slicing with DPVisor.

During the initialization of each vSDN, DPVisor installs flow entries in each related substrate switch to let it identify the packets that belong to the vSDN based their Virtual Network Headers. Specifically, the flow entries allow the substrate switch to determine a packet's tenant ID and forward it to the corresponding pipeline for further processing. Here, the pipeline refers to a sequence of flow tables that are defined and installed by the vSDN's controller. As explained in Fig. 5, upon receiving a packet, a substrate switch first determines whether it directly comes from an end-host.[1] If yes, the packet is forwarded to the pipeline defined by its tenant for

[1]In this work, we assume that the mapping between an end-host and its vSDN can be determined by the input port that the end-host connected to on the substrate switch. Therefore, if a substrate switch sees an incoming packet without the Virtual Network Header, it examines the packet's input port to find the corresponding vSDN.

processing. Otherwise, the switch deletes its Virtual Network Header and then sends it to its processing pipeline. Similarly, before outputting the packet, the switch first checks whether its next hop is an end-host or another substrate switch, and then take the actions accordingly.

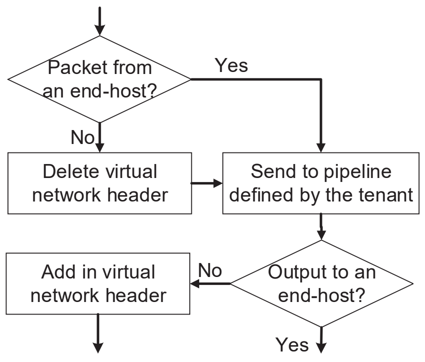
Packet from an end-host?
Yes
No
Delete virtual network header
Send to pipeline defined by the tenant
Add in virtual network header
No
Output to an end-host?
Yes

Fig. 5. Packet processing procedure in each substrate switch.

D. Failure Recovery in DPVisor

Fig. 6 explains how DPVisor realizes instant failure recovery with the distributed NVH system. Specifically, in DPVisor, when multiple NVH instances connect to the same substrate switch, one acts as the master NVH and the rest of them are slave NVHs. The master NVH has the authority to manage the substrate switch, *e.g.*, collecting the switch's status and installing flow entries/tables in it, while the slave ones also connect to the switch but cannot manage it. The substrate switches only send *Packet_In* messages to the master NVH, while the state synchronization between the master and slave NVHs is accomplished by east/west-bound communications.

Since both DPVisor and ONVisor use ONOS's working principle on NVH failure recovery, they manage the mastership of NVH instances with a Raft state machine [32], which is implemented based on the Atomix framework [33]. Specifically, in the distributed NVH systems (*i.e.*, DPVisor and ONVisor), the mastership of NVH instances changes when the master NVH instance's Raft session expires, *i.e.*, a failure happens on the instance. Here, a Raft session refers to the session between the Raft client on the master NVH instance and one of the Raft servers on the NVH instances that store the Raft state machine [33]. In normal operation, when a Raft server receives a heartbeat message from this session, it will update the session time-stamp to the current time. Hence, each server can check the session time-stamp in a polling way to determine whether the session has been expired. If it finds that the time difference between the current time and session time-stamp is longer than the preset session timeout, it determines that there is an NVH failure and will invoke the NVH failure recovery mechanism.

Note that, before the failure recovery, DPVisor actually selects the subsequent master NVH in advance to save the recovery time. Specifically, each NVH instance can connect to a substrate switch and register its interest to serve the switch in a first-in-first-out (FIFO) queue. This means that the NVH instance that first registers its interest to serve will become the master NVH instance, and the subsequent ones will be buffered in the FIFO queue in order. Then, when the master instance fails, DPVisor makes the first NVH instance in the FIFO queue as the new master instance.

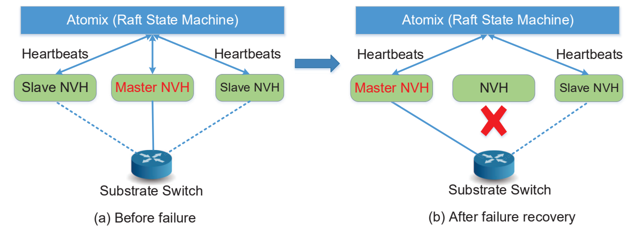

Fig. 6. Failure recovery in DPVisor.

IV. Experimental Benchmarking

In this section, we discuss the experimental evaluations of DPVisor and use the original OpenFlow-based distributed NVH system in ONOS (*i.e.*, ONVisor [17]) as the benchmark.

A. Message Processing Latency

First of all, we would like to test whether DPVisor can process messages from the substrate network as fast as ONVisor. The experimental setup is shown in Fig. 7, and to measure the message processing latency, we only deploy one NVH instance in both DPVisor and ONVisor. This is because even though DPVisor and ONVisor are distributed NVH systems, their operation principle determines that at any given time instant, only one NVH instance (*i.e.*, the master NVH) will process the messages to/from a switch. Moreover, we would like to conduct stress tests to see how well a single NVH instance can perform under message flooding. The stress tests are realized with the Cbench tool [34], which can generate a large number of *Packet_In* messages within a short period of time and measure the response time of SDN control plane. Note that, since Cbench was originally developed for the performance benchmarking of OpenFlow controllers, we extend its protocol stack to make it POF-compatible and the new version of Cbench can be found at [35]. In the experiments, we use Cbench to emulate a substrate network that has a linear topology with 10 to 100 substrate switches, and the NVH system creates vSDNs with the same topologies.

The message processing latency is defined as the time period from when a substrate switch sends out a *Packet_In* message to when the same switch receives the corresponding *Packet_Out* message from the NVH system. The Cbench tool and the NVH system both run on Linux servers (Lenovo RD540), each of which is equipped with a 2.10GHz Intel Xeon CPU and 32GB DDR3 memory. Table I compares the average message processing latencies of DPVisor and ONVisor. It can be seen that the message processing latencies of DPVisor and ONVisor are similar, and the latency of DPVisor is only slightly longer than that of ONVisor. These results confirm that even though we add in POF support in DPVisor, the implementation would not cause significant performance degradation in terms of

TABLE I
RESULTS ON AVERAGE PROCESSING LATENCY PER MESSAGE (MSEC)

# of Substrate Switches	10	20	30	40	50	60	70	80	90	100
DPVisor	0.065	0.064	0.078	0.097	0.121	0.144	0.168	0.193	0.223	0.251
ONVisor	0.065	0.063	0.074	0.091	0.114	0.132	0.159	0.187	0.209	0.244

message processing latency. Meanwhile, we notice that the latencies of DPVisor and ONVisor increase when the topology of the vSDN scales up. This is because the NVH systems spend longer time on the processing in the Virtualization Layer when the topology of the vSDN becomes larger.

B. *Message Processing Throughput*

Then, we conduct experiments to evaluate the message processing throughput of the distributed NVH systems. Here, the experimental setup is similar as that in Fig. 7, with the only differences that $\{1, 3, 5, 7\}$ NVH instances will be deployed in each distributed NVH system and the number of substrate switches is fixed as 45 (*i.e.*, so does the number of virtual switches). When multiple NVH instances are deployed in an NVH system, each instance directly manages a roughly equal number of substrate switches. For instance, if there are 3 instances in an NVH system, each instance directly manages 15 substrate switches. In the experiments, to emulate practical scenarios, each time when the vSDN controller receives a *Packet_In* message, we make it randomly choose 5 virtual switches and send *Flow_Mod* messages to them. In other words, each *Packet_In* message corresponds to a flow or 5 *Flow_Mod* messages in the experiments.

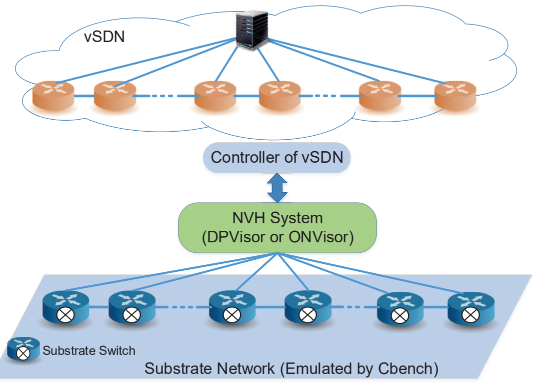

Fig. 7. Experimental setup for measuring message processing latency.

Fig. 8 shows the experimental results on the message processing throughput of DPVisor and ONVisor. It is promising to see that for both ONVisor and DPVisor, the message processing throughput increases with the number of deployed NVH instances. This verifies that the distributed NVH systems do achieve better scalability than the centralized ones. What is even more promising is that compared with ONVisor, DPVisor can achieve comparable or even higher message processing throughput. This is because when designing and implementing DPVisor, we carefully optimize the distributed operations in it. Our optimization is mainly from three aspects. Firstly, since DPVisor uses the POF protocol stack to realize network

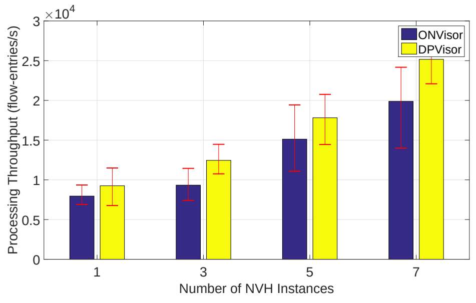

Fig. 8. Results on message processing throughput of distributed NVH systems.

virtualization, which is intrinsically more flexible than the OpenFlow protocol stack, we optimize the implementation for it in DPVisor for high efficiency. Secondly, we optimize the distributed message processing procedure in DPVisor to avoid frequent context switches of I/O threads. Specifically, to reduce the overhead due to east/west-bound communications, we design DPVisor to cache the messages to the master NVH instance on the slave ones for a short time period (*i.e.*, 10 msec) and then send them out as a batch. Finally, we separate the message dispatching thread from the I/O thread in DPVisor to avoid the interference between them, and thus the I/O thread would not be blocked easily when messages arrive at DPVisor in a relatively high speed.

C. *Failure Recovery Time*

Finally, we test how fast the distributed NVH systems can recover from an NVH instance failure. This time, since we need to send real traffic through the substrate/virtual switches, we replace the Cbench tool with our self-developed software-based POF switch [36]. The experiments use a distributed NVH system that includes $\{3, 5, 7\}$ NVH instances and connect the substrate switch with an end-host that runs Ostinato [37] for traffic flow generation. Specifically, the end-host generates 20 new flows per second and sends them to the vSDN continuously. Upon receiving the flows, the substrate switch will forward *Packet_In* messages to the distributed NVH system. During this process, we manually shut down an NVH instance in the NVH system and measure how fast it can recover from the failure. Here, we define the recovery time as the period from when the substrate switches receive the last *Packet_Out* message from the failure NVH instance to when they receive the first *Role_Request* message from the new master NVH instance. Fig. 9 shows the results on failure recover time. It can be seen that the failure recovery time of DPVisor and ONVisor is similar. Meanwhile, we notice that the failure recovery time can decrease slightly with the number of NVH instances. This is because when there are more NVH instances in the distributed NVH system, the failure can be detected slightly earlier from the statistical view point.

Note that, since both DPVisor and ONVisor leverage the default failure detection mechanism and parameter setup of ONOS, their failure recovery time is around 5 seconds. This is mainly due to the setup of heartbeat messages in ONOS, and if we shorten the period defined for heartbeat timeout, the failure recovery time can be shortened accordingly. However, using a shorter timeout period can increase the possibility of false alarms too, and thus we do not change the default setting.

V. Performance Optimization

A. Bottleneck Analysis

In the distributed NVH systems based on ONOS, the state synchronization among the NVH instances uses the Raft consensus protocol [32], which is based on the strong consistency model. This means that both the “read" (*i.e.*, NVH instances check the network status) and “write" (*i.e.*, NVH instances update the network status) operations in the NVH system have to be coordinated by the leader instance of the distributed NVH system, such that the operations will not be conducted until the leader instance can make sure that the majority of the instances are aware of them. This could lead to over-protection for the read operations and greatly limit the processing throughput of both DPVisor and ONVisor.

B. Local Cache based Scheme

In order to address the performance bottleneck discussed in the previous subsection, we design a local cache based scheme to improve the efficiency of read operations in the distributed NVH systems. Fig. 10 shows the principle of the local cache based scheme. Here, DPVisor still leverages the Raft consensus protocol [32] implemented in ONOS, which means that each NVH instance has a Raft client to access the replicated state machine in the Raft servers on certain NVH instances through a Raft session. Note that, among the Raft servers, Raft protocol first determines a leader through leader election, and then the remaining servers just become its followers. We can refer to the leader and all the followers as the Raft cluster.

As shown in Fig. 10, all the write operations are fed into the Raft cluster by the Raft clients. When the leader receives these operations, it writes Raft logs to its local virtual network data store, sends the logs to the followers in batch, and then responds to the Raft client about the fact that the operations have been committed. When an operation has been committed, the result will be published by the Raft server that the client is connected to with a monotonic sequence number. Then, the client will reorder the results published from the Raft servers according to the sequence number, and send them to the local cache one by one. In the case that a published result is lost, the client will contact the server for republishing the result. Hence, the local cache will see the updated results in the correct sequence. Fig. 10 also shows that the local cache based scheme lets write operations be executed through the Raft client, while for the read operations, they are executed directly on the local cache to bypass expensive remote calls to the leader. To this end, we can see that our local cache based scheme can achieve a semantic of linearizable write [32, 38] and sequential read, *i.e.*, maintaining the correct update order of the virtual network store. Therefore, with the local cache based scheme, the distributed NVH system still uses the strong consistency model for write operations, but bypasses it for all the read operations. Note that, during network operation, read operations are conducted much more frequently than write operations, and thus our design can greatly improve the processing throughput of the NVH system.

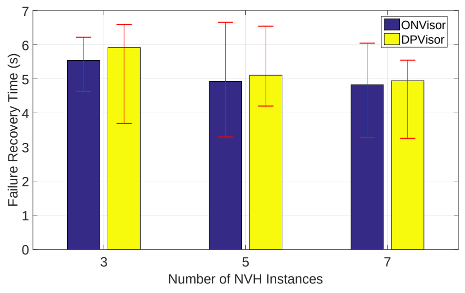

Fig. 9. Results on failure recovery time of distributed NVH systems.

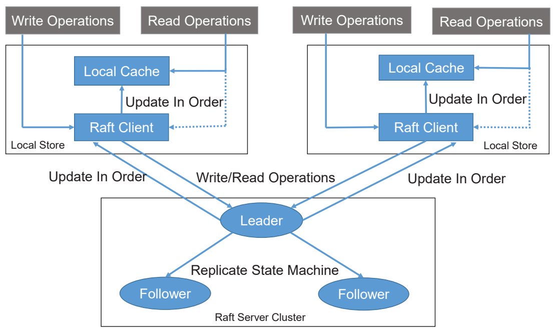

Fig. 10. Principle of the local cache based scheme.

To verify the effectiveness of the local cache based scheme, we implement it in DPVisor and ONVisor and conduct experiments to measure their message processing throughput with the setup in Section IV-B. Here, we also use the distributed NVH systems that utilize the eventual consistency model for state synchronization as the benchmarks. Note that, the eventual consistency model can completely relax the restrictions of the strong consistency model, in the way that each NVH instance just performs read or write operations at will and then updates the results to the other instances in the best effort manner. However, the eventual consistency model can sacrifice data consistency for data availability, which will not happen when using the strong consistency model or the local cache based scheme. Fig. 11 shows the experimental results on the message processing throughput. It can be seen that the NVH system with the local cache based scheme achieves a comparable message processing throughput as that of the system with the eventual consistency model, while their throughput is significantly higher than that of the system with the strong consistency model, *i.e.*, around 4 times higher when the number of NVH instances is 7. The results of DPVisor are plotted in Fig. 12, which indicates the similar trend. Hence, the results verify that the processing throughput of distributed NVH system can be greatly improved with our local cache based scheme (*i.e.*, for both DPVisor and ONVisor).

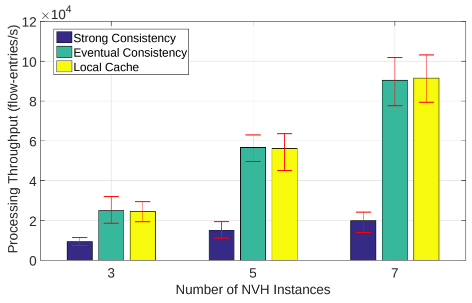

Fig. 11. Results on message processing throughput of ONVisor.

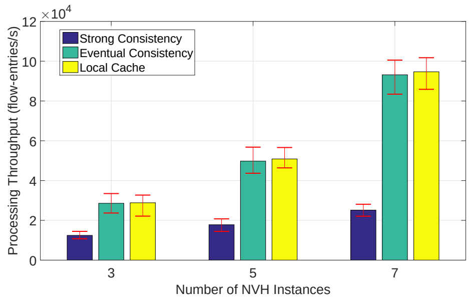

Fig. 12. Results on message processing throughput of DPVisor.

Note that, the strong consistency model makes sure that the read operations on all the NVH instances always obtain state updates in the same order, which cannot be achieved by the eventual consistency model. However, keeping the correct update order in all the NVH instances is essential to realize effective vSDN slicing. For example, a newly-added virtual port should only be seen by the NVN instances after the virtual switch that the port belongs to has been added in them. Therefore, we conduct experiments to compare DPVisor's performance on keeping the correct update order, when being implemented with the local cache based scheme and the eventual consistency model. In the experiments, we choose the number of NVH instances in DPVisor within $\{3, 5, 7\}$, update the distributed virtual network store on one NVH instance continuously, and record all the updates on other NVH instances. Then, we calculate the percentage of the updates whose orders are the same as the original ones on each NVH instance, *i.e.*, the percentage of correct update orders, and plot the minimum results in DPVisor in Fig. 13. The results can be used as a performance metric to measure the data consistency in the distributed NVH system. It can be seen that the local cache based scheme always ensures 100% correct update order, but the eventual consistency model cannot even achieve over 5% correct update order. These results confirm the advantage of the local cache based scheme on maintaining data consistency in the distributed NVH system.

Compared with the strong consistency model, both the local cache based scheme and eventual consistency model

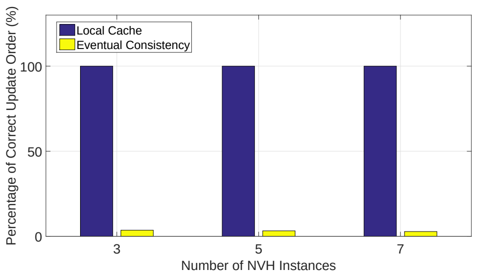

Fig. 13. Percentage of correct update order in DPVisor.

cannot guarantee that all the NVH instances see the newest network status in read operations. Hence, we perform more experiments to measure the performance of DPVisor on data inconsistency. In the experiments, the distributed NVH system still includes $\{3, 5, 7\}$ NVH instances, and we update the distributed virtual network store on one NVH instance once per second and measure the synchronization time among all the NVH instances. Since during the synchronization time, the NVH instances cannot process control messages with the newest network status, we use the synchronization time to derive the number of control messages that are processed by DPVisor with outdated network status and plot the results in Fig. 14. The results indicate that the percentage of data inconsistency of the local cache based scheme is below 0.7% and much smaller than that of the eventual consistency model. This verifies that the local cache based scheme can maintain the data consistency among NVH instances well.

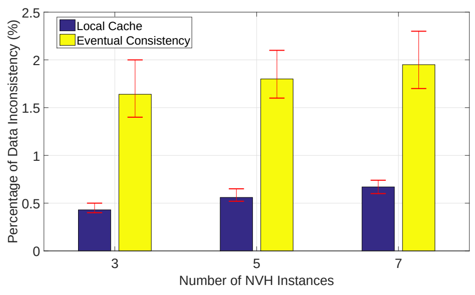

Fig. 14. Percentage of messages processed with outdated status.

VI. Conclusions

We designed and implemented a distributed NVH system based on ONOS, namely, DPVisor, to realize highly-available, scalable and protocol-independent vSDN slicing. We first expanded the NVH module in ONOS to add in the support of POF for enabling protocol-independent vSDN slicing. Then, we transformed the centralized NVH system into a distributed one by creating distributed NVH instances and letting them use east/west-bound communications to maintain a logically centralized view of the substrate network and vSDNs. The experimental comparisons indicated that DPVisor achieved comparable performance as ONVisor (*i.e.*, an OpenFlow-based ONOS benchmark), in terms of the message processing latency, message processing throughput, and failure recovery time. In order to optimize the performance of distributed NVH systems, we also carefully adjusted the consistency model used in the state synchronization of NVH instances and proposed a local cache based scheme to balance the tradeoff between the consistency and availability of network status. Our experimental results confirmed that the message processing throughput of distributed NVH systems could be greatly improved (*i.e.*, for both DPVisor and ONVisor), while the data consistency among NVH instances was still maintained well.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC Project 61371117, the Key Project of the CAS (QYZDY-SSW-JSC003), the NGBWMCN Key Project under Grant No. 2017ZX03001019-004, and the Strategic Priority Research Program of the CAS (XDA06011202).

REFERENCES

[1] L. Gong, Y. Wen, Z. Zhu, and T. Lee, "Toward profit-seeking virtual network embedding algorithm via global resource capacity," in *Proc. of INFOCOM 2014*, pp. 1–9, Apr. 2014.
[2] T. Anderson *et al.*, "Overcoming the Internet impasse through virtualization," *IEEE Computer*, vol. 38, pp. 34–41, Apr. 2005.
[3] L. Gong and Z. Zhu, "Virtual optical network embedding (VONE) over elastic optical networks," *J. Lightw. Technol.*, vol. 32, pp. 450–460, Feb. 2014.
[4] L. Gong, H. Jiang, Y. Wang, and Z. Zhu, "Novel location-constrained virtual network embedding (LC-VNE) algorithms towards integrated node and link mapping," *IEEE/ACM Trans. Netw.*, vol. 24, pp. 3648–3661, Dec. 2016.
[5] Z. Zhu *et al.*, "Build to tenants' requirements: On-demand application-driven vSD-EON slicing," *J. Opt. Commun. Netw.*, vol. 10, pp. A206–A215, Feb. 2018.
[6] A. Al-Shabibi *et al.*, "OpenVirteX: Make your virtual SDNs programmable," in *Proc. of ACM HotSDN 2014*, pp. 25–30, Aug. 2014.
[7] J. Yin *et al.*, "On-demand and reliable vSD-EON provisioning with correlated data and control plane embedding," in *Proc. of GLOBECOM 2016*, pp. 1–6, Dec. 2016.
[8] S. Li *et al.*, "SR-PVX: A source routing based network virtualization hypervisor to enable POF-FIS programmability in vSDNs," *IEEE Access*, vol. 5, pp. 7659–7666, 2017.
[9] J. Yin *et al.*, "Experimental demonstration of building and operating QoS-aware survivable vSD-EONs with transparent resiliency," *Opt. Express*, vol. 25, pp. 15 468–15 480, 2017.
[10] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, "Survey on network virtualization hypervisors for software defined networking," *IEEE Commun. Surveys Tuts.*, vol. 18, pp. 655–685, First Quarter 2016.
[11] R. Sherwood *et al.*, "FlowVisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep.*, pp. 1–13, 2009.
[12] B. Zhao, X. Chen, J. Zhu, and Z. Zhu, "Survivable control plane establishment with live control service backup and migration in SD-EONs," *J. Opt. Commun. Netw.*, vol. 8, pp. 371–381, Jun. 2016.
[13] R. Govindan *et al.*, "Evolve or die: High-availability design principles drawn from Google's network infrastructure," in *Proc. of ACM SIGCOMM 2016*, pp. 58–72, Aug. 2016.
[14] X. Chen *et al.*, "Leveraging master-slave openflow controller arrangement to improve control plane resiliency in SD-EONs," *Opt. Express*, vol. 23, pp. 7550–7558, Mar. 2015.
[15] P. Berde *et al.*, "ONOS: Towards an Open, Distributed SDN OS," in *Proc. of ACM HotSDN 2014*, pp. 1–6, Aug. 2014.
[16] S. Li *et al.*, "Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability," *IEEE Netw.*, vol. 31, pp. 12–20, Mar./Apr. 2017.
[17] ONOS network virtualization. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Network+Virtualization
[18] Y. Han *et al.*, "ONVisor: Towards a scalable and flexible SDN-based network virtualization platform on ONOS," *Int. J. Netw. Manag., in Press*, 2017.
[19] H. Huang *et al.*, "Embedding virtual software-defined networks over distributed hypervisors for vDC formulation," in *Proc. of ICC 2017*, pp. 1–6, May 2017.
[20] Z. Bozakov and P. Papadimitriou, "AutoSlice: Automated and scalable slicing for software-defined networks," in *Proc. of CoNEXT Student 2012*, pp. 3–4, Dec. 2012.
[21] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Comput.*, vol. 17, pp. 20–27, Mar. 2013.
[22] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.
[23] Protocol Independent Forwarding. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/OF-PI__A_Protocol_Independent_Layer_for_OpenFlow_v1-1.pdf
[24] S. Li *et al.*, "Improving SDN scalability with protocol-oblivious source routing: A system-level study," *IEEE Trans. Netw. Serv. Manag., in Press*, 2017.
[25] D. Hu *et al.*, "Design and demonstration of SDN-based flexible flow converging with protocol-oblivious forwarding (POF)," in *Proc. of GLOBECOM 2015*, pp. 1–6, Dec. 2015.
[26] S. Li, D. Hu, W. Fang, and Z. Zhu, "Source routing with protocol-oblivious forwarding (POF) to enable efficient e-health data transfers," in *Proc. of ICC 2016*, pp. 1–6, Jun. 2016.
[27] D. Hu *et al.*, "Flexible flow converging: A systematic case study on forwarding plane programmability of protocol-oblivious forwarding (POF)," *IEEE Access*, vol. 4, pp. 4707–4719, 2016.
[28] Q. Sun, Y. Xue, S. Li, and Z. Zhu, "Design and demonstration of high-throughput protocol oblivious packet forwarding to support software-defined vehicular networks," *IEEE Access*, vol. 5, pp. 24 004–24 011, 2017.
[29] K. Han *et al.*, "Leveraging protocol-oblivious forwarding (POF) to realize NFV-assisted mobility management," in *Proc. of GLOBECOM 2017*, pp. 1–6, Dec. 2017.
[30] S. Zhao *et al.*, "Make Big Data applications more reliable: Hitless vSDN migration to avoid TCAM depletion," in *Proc. of ICC 2018*, pp. 1–6, May 2018.
[31] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*. Addison Wesley Longman, 2000.
[32] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of USENIX ATC 2014*, pp. 305–319, Jul. 2014.
[33] Atomix. [Online]. Available: http://atomix.io/atomix
[34] OpenFlow Cbench Tool. [Online]. Available: https://github.com/mininet/oflops/tree/master/cbench
[35] POF Cbench Tool. [Online]. Available: https://github.com/USTC-INFINITELAB/pof-cbench
[36] POF Software Switch. [Online]. Available: https://github.com/USTC-INFINITELAB/POFSwitch
[37] Ostinato. [Online]. Available: http://ostinato.org/
[38] M. Herlihy and J. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Progr. Lang. Sys.*, vol. 12, pp. 463–492, Jul. 1990.