2016 IEEE 24th International Conference on Network Protocols (ICNP)
Workshop on COntrol, Operation and appLication in SDN protocols (CoolSDN 2016)

NeSMA: Enabling Network-Level State-Aware Applications in SDN

Chen Sun[1], Jun Bi[1], Hongxin Hu[2], Zhilong Zheng[1]
[1]Institute for Network Sciences and Cyberspace, Tsinghua University
[1]Department of Computer Science, Tsinghua University
[1]Tsinghua National Laboratory for Information Science and Technology (TNList)
[2]Clemson University
c-sun14@mails.tsinghua.edu.cn junbi@tsinghua.edu.cn hongxih@clemson.edu zhengzl15@mails.tsinghua.edu.cn

***Abstract*—As the de facto data plane technique of Software-Defined Networking (SDN), OpenFlow introduces significant programmability to enable innovative network applications. However, the simple OpenFlow data plane only maintains flow-level counters and lacks an efficient mechanism to manage network-level states, which limits its support for advanced state-aware applications. Regularly pulling whole state information from the data plane to the controller might incur untimely response to important network-level states such as CPU exhaustion, switch overload, etc and cause unnecessary traffic. To address above challenges, we introduce a novel Network-level State Management Architecture (NeSMA) to efficiently support advanced network-level state-aware applications by exploiting the opportunity of SDN central control. The data plane could be configured to check state regularly and report to the controller when triggered by state transitions. We design both sequential and parallel composition methods to deal with complex network-level states in NeSMA. To demonstrate the feasibility of our approach, we implement a software prototype of NeSMA, based on which we develop a data-center flow scheduling application. Experimental results show that NeSMA can process network-level states with low network resource consumption and high scalability without compromising packet forwarding efficiency.**

I. Introduction

Software-Defined Networking (SDN) is an emerging network architecture that provides unprecedented programmability, automation, and network control by decoupling the control plane and the data plane. As a representative technique of SDN, OpenFlow [13] introduces a "match-action" paradigm for the SDN data plane where programmers could specify a flow through a header matching rule along with processing actions applied to the matched packets. OpenFlow switches remain simple and are only in charge of forwarding, while all intelligence is placed at the controller side.

Although the programmability of OpenFlow significantly helps manage and process network flows and is effective for many applications on top of the controller, the simplicity of OpenFlow abstraction also brings great challenges in building advanced state-aware network applications such as load balancing [11], fast reroute [20], flow scheduling [7], etc. The necessity to add states into OpenFlow has been recently recognized by the research community [8], [9], [12], [14], [21], [22]. In most of exiting work, the researchers claim that current SDN switches cannot support complex stateful processing and define states of network in a *local, flow-level* scope to support some stateful network applications. However, those existing approaches are only applicable for designing network applications, such as MAC learning and port knocking, which simply need *local* states of flows on a *single* switch.

The term "state" in networking can be defined as historical information that may affect future packet processing policies. We classify "state" into *flow-level* and *network-level* according to the scale it represents. *Flow-level* states refer to local states associated with traffic flows that are collected inside one switch and will only affect the forwarding policy in the same switch. Typical flow-level states include flow TCP connection state, packet counter of flows, FTP control channel state, etc. For instance, a stateful firewall keeps tracking TCP connection states of each flow and determines packet handling process based on stateful policies. On the other hand, *network-level* states refer to states that exist in multiple switches and may influence the packet processing policies on other switches. For example, in a flow scheduling application, if a particular switch is overloaded, other switches that are transmitting traffic to this switch should be configured to steer traffic away.

However, our study reveals that there are great challenges in building network-level state-aware applications in SDN. Firstly, exiting OpenFlow data plane can only maintain flow-level counters and lacks an effective mechanism to gather other useful network-level states, such as hardware TCAM usage, and node processing latency, that are critical for building many state-aware applications. Secondly, there is no efficient way to quickly react to network-level state transitions. Regularly pulling the whole state information from the data plane to the controller might incur untimely response to some network-level states such as CPU exhaustion, switch overload, etc. Some protocols for SDN including NETCONF [10], OVSDB [16] and OF-Config are capable of monitoring and configuring switches during runtime. However, they could not monitor some resource and performance states as mentioned above, and still suffer from untimely response caused by controller initiated control actions.

To address above challenges, we introduce an innovative

978-1-5090-3281-5/16/$31.00 ©2016 IEEE

Network-level State Management Architecture (NeSMA) to support complex state-aware applications in SDN with little resource consumption and great scalability. We add a *Network State Processor* in the data plane to monitor states from outside the OpenFlow flow table pipeline, exerting no influence on forwarding performance. We adopt a *triggered reporting* mechanism in the data plane to report to the controller on state transition. We also propose sequential and parallel compositions of network level states in NeSMA. Programmers can then construct sophisticated network-level state-aware applications through the two state composing mechanisms.

The paper makes the following contributions:

- We propose a novel network-level state management architecture, NeSMA, to support network-level state-aware applications in SDN. We design control plane and data plane enhancements for NeSMA, along with corresponding APIs.
- We propose both sequential and parallel composition approaches to process complex network-level states.
- We implemented a software prototype of NeSMA and developed a data-center flow scheduling application. Experimental results show that NeSMA does not compromise data plane forwarding performance and can support network-level states in SDN with little network resource consumption and great scalability.

The rest of this paper is organized as follows. Section II elaborates the detailed design of each module of NeSMA. Section III describes two state composition methods supported by NeSMA, including sequential and parallel compositions. Section IV introduces the implementation and evaluation of NeSMA. Section V lists some related works. We conclude this paper in Section VI.

II. Design

A. Intuitions and Design Overview

We observe that network-level states have some unique characteristics compared to flow-level states. (1) *Few Categories with High Reusability:* A determined set of network-level states need to be gathered including throughput, CPU usage, memory usage, etc. We could ignore the complexity of various applications and only focus on limited types of states. (2) *Involving multiple nodes:* This is a major difference from flow-level states. Network-level states in one node should be transmitted to invoke policy changes on other nodes. Thus, an efficient delivery of states is required to achieve both timeliness and resource efficiency. (3) *High Complexity:* One application might need to monitor multiple states in sequential or parallel. For instance, a load balancing application may react differently to a 70%, 80% or 90% loaded switch. It needs to monitor those four grades of throughput at the same time.

Meanwhile, SDN naturally brings great opportunity for supporting network-level states. The centralized controller is capable of timely sensing and controlling data plane devices. Thus, network-level states can simply be sent to the controller for applications to affect other switches. Moreover, network intelligence is centralized in the controller. Applications running on top of the controller can identify exactly the source and influence target of states, providing opportunity for efficient management of and reaction to network-level states.

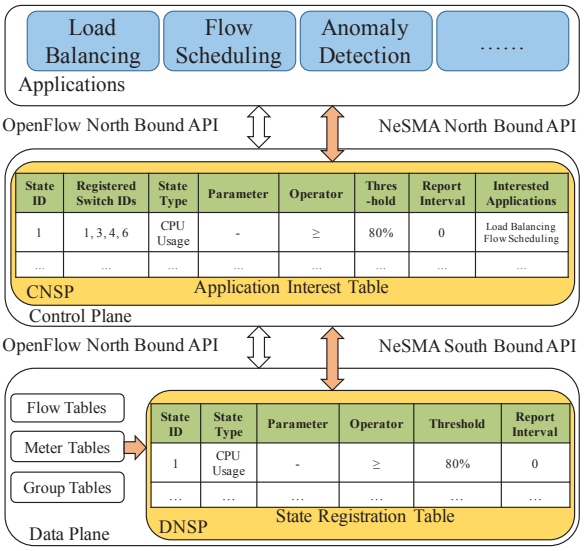


Fig. 1: NeSMA Architecture

Therefore, the design philosophy of NeSMA is to extend OpenFlow to support timely network-level state monitoring and response, while keeping the forwarding pipeline fast and simple. A strawman solution to support network-level state monitoring is for switches to inform each other directly about states. However, a switch itself does not know which switches care about its state. State flooding may cause unnecessary traffic and extra policies to forward states among switches. Besides, in SDN each switch does not have enough intelligence to react to network-level states. In NeSMA, we propose a "trigger report" mode where the controller will *configure* each switch about what states to monitor. The switch checks whether the state is detected. A certain state appearance could trigger the switch to report to the controller. The controller will deliver the state to associated applications. Finally, applications will generate new policies and distribute them to relevant switches. For a load balancing application, the controller informs a switch to report when its throughput reaches 80% of its capacity. The switch checks its throughput regularly and report to the controller when the threshold is reached. The load balancing application will distribute traffic steering policies to associated switches to alleviate the hotspot. In order to satisfy the requirements of some applications, we also support two other modes including switch regularly reporting state to the controller, and the controller actively pulling states from switches, which will be elaborated in the following sections.

We design three fundamental components in NeSMA (Fig. 1) to support network-level state-aware applications. We design a processing unit in the software control path of SDN switches named *Data Plane Network State Processor (DNSP)*. We maintain a *State Registration Table (SRT)* in DNSP, which records what states the switch should monitor and report. In

TABLE I: State Types Supported in NeSMA

State Type	State Collection Method
States in OF meter table	Fetch state from OpenFlow meter table, including per flow table packet lookups and matches, per flow entry packets, per port packets, per queue packets, etc.
CPU Usage	Perform system calls to get system CPU usage.
TCAM Usage	This state can only be fetched for hardware switches. DNSP should maintain a table entry number and be updated each time a flow entry is added or removed from the hardware switch.
SRAM Usage	Perform system calls to get system SRAM usage.
Switch Availability	This state is generated by the controller when sending *Keep Alive* messages to switches.

the control plane, we design a *Control Plane Network State Processor (CNSP)*, in which an *Application Interest Table (AIT)* is maintained to record the applications' interests on states. In addition, we design south-bound and north-bound APIs to enable NeSMA.

B. State Monitoring and Reporting Modes

We design three mechanisms for switches to monitor states and report to the controller.

- *Triggered Reporting*: In this mode, the controller configures the switches to report when certain states are detected. This method ensures timely report and response, saves network bandwidth and preserves scalability. Besides, another benefit of triggered reporting mode is to relieve the controller from examining various states against various thresholds proposed by applications, which is a heavy task for the controller and would occupy controller capacity.
- *Regular Reporting*: This mode is used when an application tries to maintain historical states in a period for statistical survey, classification, etc [17], [19]. Compared with traditional OpenFlow, the controller can be offloaded from regularly pulling states from the data plane, which doubles the transmission latency and bandwidth cost.
- *Proactive Pulling*: This mode is used when the application requires an immediate report of current states. For instance, if a traffic anomaly detection applications detects an attack on one switch, it needs to acknowledge immediately whether the same attack happens on other switches. Thus, the controller should be capable of pulling states from the data plane.

In NeSMA, these three mechanisms are supported through both control plane and data plane extensions.

C. Data Plane Enhancement

1) State Types: The control plane and the data plane should agree on a set of states that the switches can gather in the data plane. Types and state collection methods are listed in Table I.

2) Data Plane Network State Processor: Data Plane Network State Processor (DNSP) records states to monitor and report to the controller. It can receive and process controller configurations, gather states, and report on state transitions.

DNSP is placed in the software control path of SDN hardware or software switches, outside the packet forwarding pipeline. If we fix DNSP inside the datapath, it will be difficult to monitor states including CPU usage, SRAM usages and other states that reside out of the data path. Besides, DNSP outside the data path will not interfere packet processing. Forwarding performance will therefore not be compromised.

For *Triggered Report* mode, DNSP collects states with a *Default Interval*, which is maintained in DNSP and can be modified by the controller. If an application requires a more timely report of state appearance, the controller could decrease the *Default Interval* to enable more intense state monitoring.

3) State Registration Table: A State Registration Table (SRT) is used to keep configurations from the controller about what states to collect. The structure of SRT is shown in Fig. 2.

State ID	State Type	Parameter	Operator	Threshold	Report Interval	Report Detail
1	CPU Usage	-	$\geq$	80%	0	-
2	Port Packet Number	Port ID = 1	$<$	10,000	0	Packet IP addresses
3	Queue Length	Queue ID = 1	-	-	30	-

Fig. 2: SRT Structure

- *State ID*: This field is uniquely assigned by the controller. When a certain state is detected, the switch will include its ID in the report to mark the transition of this state.
- *State Type*: State Type field is encoded as an enumerate parameter for states listed in Table I.
- *Parameter*: This field is to assign parameters for different states, such as *Port ID* for *Per Port Packet Number* state.
- *Operator*: This field maintains simple comparison operators including $<, >, \leq, \geq, =, \neq$.
- *Threshold*: This field notes the threshold to report a certain state detection to the controller.
- *Report Interval*: Each entry could specify a *Report Interval*. If this field has a non-zero value, it will enable the *Regular Report* mode, and *Triggered Report* mode otherwise.
- *Report Detail*: This field is assigned to indicate the content to report to the controller. For instance, if a switch is configured to detect elephant flows, the switch could report the appearance of an elephant flow as well as the IP addresses of this flow. The report details for each states could be flexibly assigned by the controller according to application requirements.

If $State\ Type(Parameter) - Operator - Threshold$ is *TRUE* or the timer reaches the $Report\ Interval$, the state should be reported to the controller. SRT entries can be maintained in SRAM instead of TCAM since exact matches are performed on SRT entries for state monitoring. SRT entries can only be added or removed by the controller.

D. Control Plane Enhancement

1) Control Plane Network State Processor: Control Plane Network State Processor (CNSP) maintains the applications' interests on states in Application Interest Table (AIT). It numbers states uniquely and records them in AIT. It provides interfaces for applications to register or unregister interests. CNSP is able to change the *Default Interval* in the data plane according to applications' requirements for faster or slower state monitoring. As mentioned in *Proactively Pulling* mechanism, CNSP should support directly pulling states from the data plane.

2) Application Interest Table (AIT): It is used to maintain each applications' interests on states as shown in Fig. 3.

State ID	Registered Switch IDs	State Type	Parameter	Operator	Thres -hold	Report Interval	Report Detail	Interested Applications
1	1, 3, 4, 6	CPU Usage	-	≥	80%	0	-	Load Balancing Flow Scheduling
2	1, 3, 5	Port Packet Number	Port ID = 1	<	10,000	0	Packet IP addresses	Load Balancing Anomaly Detection
3	2, 7, 8, 9, 10	Queue Length	Queue ID = 1	-	-	30	-	Flow Scheduling

Fig. 3: AIT Structure

- *State ID*: The controller should *uniquely* number states required by applications. *State ID* is a consensus mark between the application, control and data plane managed by controller.
- *Registered Switch IDs*: This field records which switches should monitor the state in this field. This field can be set to empty if all switches need to gather this state.
- *Interested Applications*: This field records the applications that are interested in the state of this entry. Such aggregation exploits the high reusability of network-level states and will decrease table size, and shorten the time for table lookup and state delivery to applications.

E. NeSMA APIs

In order to build network-level state-aware applications in SDN, we design north-bound API on top of the CNSP and south-bound API between the CNSP and DNSP. North-bound API is mainly used for applications to register or unregister their interests. South-bound API is mainly used for communications between CNSP and DNSP, including installing or removing SRT entries in the DNSP, configuring *Default Interval*, and directly pulling states from the data plane. API designs are elaborated as follows.

1) South-Bound API: The south-bound API mainly includes five key interfaces.

- *Default Interval Configuration*: This function is used to configure state reporting interval in DNSP by CNSP to speed up or slow down state monitoring and checking frequency.
- *SRT Entry Addition*: This function is used to add entries into SRT by CNSP during runtime. Applications call this interface to enable some switches to monitor interesting states.
- *SRT Entry Removal*: This function is used to remove entries from switch SRT by CNSP if the state should no longer be monitored by the switch.
- *State Pulling*: This interface is used to pull states from SRT by CNSP during runtime. Applications will call this function when they need to acknowledge current states. Besides, CNSP could call this function regularly to *keep-alive* the switches.
- *State Report*: This function is invoked by DNSP to report state detection or transition to CNSP. This is the only reaction of switches proactively or reactively.

2) North-Bound API: The north-bound API includes the following major functions.

- *Interest Registration*: Applications call this interface to register an interest for states into AIT. CNSP will decide whether to aggregate this interest into an existing entry or append a new entry to AIT.
- *Interest Removal*: An applications calls this interface to remove its interest on a state. If there are still other applications watching this state, CNSP will simply remove this application from *Interested Applications*. If not, CNSP will remove state monitoring on all relevant switches and delete the AIT entry.
- *State Pulling*: Applications call this interface to immediately get states from the data plane.
- *State Delivery*: This interface is called by CNSP to deliver a state to registered applications upon receiving a state report.

III. Network-level State Composition

A state-aware application often needs to monitor multiple states (e.g Packet rate, CPU usage and Memory Usage) on the same switch. NeSMA offers two simple network-level state composition methods for programmers to support the monitoring of multiple states in sequential or in parallel.

A. Sequential Composition

Some applications require one state to be monitored after the other state is detected. Take data plane flow scheduling as an example. Hedera [7] detects elephant flows by letting the controller pull the flow counters on switches.

However, pulling is a very heavy task. In NeSMA, switches are able to judge elephant flows and actively report to the controller as long as the controller tells them the threshold of flow counters. We divide this application into two stages. (1) Network congestion detection. The application can decide a threshold of packet rate of switches, over which a switch is considered as congested. Once the controller receives network congestion report, it will inform the application. (2) Elephant flow detection. Once the network is congested, the controller should configure new SRT entries to detect elephant flows. Switches will monitor the counter of all flows and report a flow with a larger counter than the threshold. The pseudo code of state composing of flow scheduling in NeSMA is shown as follows.

```
/* Register interest about network congestion state */
Function initialize ()
    register_application_interest (
     state_type = CNSP.THROUGHPUT,
     state_operator = CNSP.GREATER_THAN,
     state_threshold = 80% * SWITCH.CAPACITY,
     report_interval = 0, /*Trigger Report mode.*/);

/*Receive and parse states to different handlers.*/
Function state_parser ( state_id , state_appear , state_detail )
  switch ( state_id ): {
    case congestion_state_id :
      if ( state_appear == true)
        /*Congested, enable monitoring elephant flows.*/
        enable_monitor_elephant_flow( state_detail );
      else
        /*Not congested, no longer monitor elephant flows.*/
        remove_monitor_elephant_flow( state_detail );
      break;
    case elephant_flow_state_id :
      if ( state_appear == true)
        /*Elephant flow detected.*/
        handle_elephant_flow( state_detail );
      break;
  }

/* Register interest about elephant flow state */
Function enable_monitor_elephant_flow( state_detail ) {
   elephant_flow_state_id = CNSP. register_application_interest
      (
     state_type = CNSP.FLOW_PACKET_NUMBER,
     state_operator = CNSP.GREATER_EQUAL,
     state_threshold = 10000,
     report_interval = 0, /*Trigger Report mode.*/);
}
```

B. Parallel Composition

Some applications need to monitor multiple states or different grades of the same state at the same time. In statistical anomaly detection [15], the application needs to maintain ordinal measure, such CPU usage, memory usage, packet rate, to generate a periodical profile. It needs to monitor multiple states altogether. Thus, a parallel composition of the states is required to support such applications.

We enable parallel composition by registering multiple interests to CNSP at the same time. During initialization, the application should register interests of all three states concerned to CNSP. No matter which state is detected, the application needs to recalculate local profile and make judgments about the appearance of anomaly. Pseudo code of statistical anomaly detection application is given as follows.

```
/* Register interest about all interesting states */
Function initialize ()
    register_application_interest (
     state_type = CNSP.CPU_USAGE,
     state_operator = CNSP.GREATER_THAN,
     state_threshold = 60%,
     report_interval = 30, /*Regular Report mode.*/);
    register_application_interest (
     state_type = CNSP.MEMORY_USAGE,
     state_operator = CNSP.GREATER_THAN,
     state_threshold = 60%,
     report_interval = 30, /*Regular Report mode.*/);
    register_application_interest (
     state_type = CNSP.THROUGHPUT,
     state_operator = CNSP.GREATER_THAN,
     state_threshold = 60% * SWITCH.CAPACITY,
     report_interval = 30, /*Regular Report mode.*/);

/*Receive states and update local profile .*/
Function state_parser ( state_id , state_appear , state_detail )
  if ( state_appear == true) {
    update_current_profile ( state_id , state_detail );
    break;
  }
```

IV. Implementation and Evaluation

A. Implementation

We extended Floodlight [5] to be NeSMA controller. We developed the CNSP as a module application in Floodlight (1.2K LoC) and designed AIT inside CNSP to record application interests on states. In addition, we extended Floodlight to support south-bound and north-bound APIs of NeSMA. Open vSwitch (OVS) [4] was extended to support NeSMA data plane. We appended DNSP module into OVS which maintains SRT table and collect states according to SRT entries (2.5K LoC). Besides, we also implemented south bound interfaces in DNSP to receive configurations and report states. We implemented a flow scheduling application based on NeSMA prototype, which monitors data plane congestion and switch availability in parallel, and steers traffic accordingly. We realized this application using the triggered reporting mode.

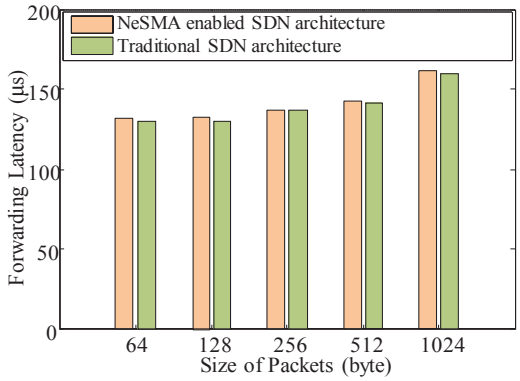
We conduct our experiments based on DELL R720 servers, the CPU of which is Xeon E5-2609 (2.4 GHz). We use IXIA [2] tester to generate packets for our experiments.

B. Evaluation

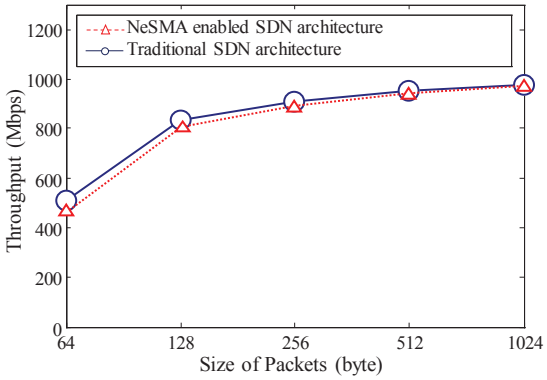
1) NeSMA's Impact on Forwarding Performance: In our design, DNSP is placed outside the packet forwarding pipeline. Theoretically, packet forwarding latency and throughput performance will not be compromised by enabling NeSMA in SDN architecture. State retrieval for CPU and memory usage requires a simple system call for software switches. In order to fetch states in the OpenFlow meter table, we construct OpenFlow messages from DNSP and send them to OVS, which could occupy little link bandwidth since all messages are transmitted inside the switches.

We prove this by enabling and disabling simple flow scheduling application and measuring forwarding latency and throughput respectively. We run our experiment with the controller on one server and a switch on the other server. The two servers are directly connected. We use IXIA packet generator to send packets whose size range from 64 bytes to 1024 bytes, and evaluated the forwarding latency and throughput with or without NeSMA enhancement. As shown in Fig. 4, adding NeSMA in SDN will only compromise forwarding latency and throughput a little bit, due to the CPU cost of data plane state collection, message formation and state reporting in DNSP. This demonstrates the feasibility of implementing NeSMA in SDN without much affecting data path forwarding performance.

We further prove the low throughput overhead of NeSMA in a Mininet [3] simulation environment based on a real-world network topology derived from the Stanford backbone network [1]. We selected a three-hop forwarding path.

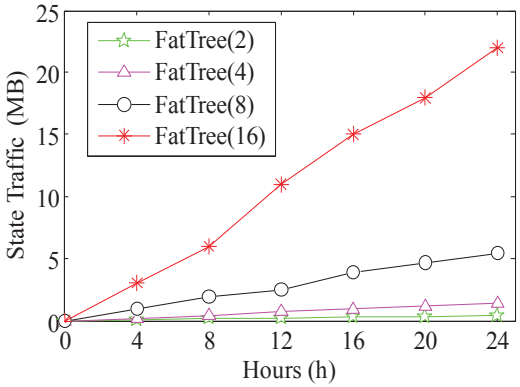


(a) Forwarding Latency

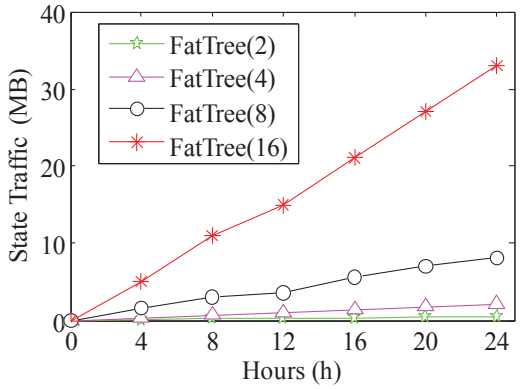


(b) Throughput

Fig. 4: Packet forwarding performance with/without NeSMA



(a) Regular reporting by switches



(b) Proactive pulling by the controller

Fig. 5: State related traffic amount in 24 hours

For packets of 1024 bytes, the throughput is 9.5Gbps with NeSMA enhancement, while the throughput without NeSMA is 9.6Gbps. This illustrates a minimal overhead introduced by NeSMA.

2) State Reaction Latency: Based on our implementation of flow scheduling, we evaluated the latency between the time when a state is detected and the time when reactive policies are installed in the data plane. We use Mininet to simulate the Stanford backbone network architecture from Hassel. We overload one of the switch for the controller to detect switch congestion state. We configure the switch to check states once every 5 seconds and adopt the *Triggered Reporting* Mode. In our experiments, the average latency is approximately $142\mu s$ between state detection and policy installment, which is rather fast and efficient for state reaction.

In the Mininet simulation, each switch is directly connected to the controller, which means the transmission distance is only one hop. This might not be the case in real world network topology. If the control packets are transmitted in-band, the distance between the controller and switch is always more than one hop. Thus, we additionally implement Fat-Tree [6] topology and configure all switches to report their congestion states to the controller. We connect all core switches to the controller, leaving the control packets of other switches transmitted in-band. Thus, the distance from core layer, aggregate layer and edge layer switches to the controller is $1, 2, 3$ respectively. In this structure, we can calculate that the average hops from the switches to the controller is:

$$average\_distance = \frac{1 * (k^2/4) + 2 * (k^2/2) + 3 * (k^2/2)}{h^2/4 + k^2/2 + k^2/2}$$
$$= 11/5 \approx 2$$

Above result shows that in Fat-Tree topology, average distance from the controller to a switch is approximately 2 hops, which will add only a little bit to the $142\mu s$ latency.

3) Network Resource Occupation and Scalability: The scalability of NeSMA comes from the little resource occupation of state transmitting. According to NeSMA design, the size of an NeSMA state report message is 25 bytes. We configure all switches to report state in the relatively resource-consuming *Regular Report* mode, and *Controller pulling* mode once every 30 seconds. We use Mininet to simulate Fat-Tree Topology with different number of switches and run the simulation for 24 hours. Experimental results are shown in Fig. 5. Only 23M bytes of state traffic is generated during 24 hours for a FatTree (16) network with 320 switches in *Regular Report* mode, and 32M bytes of traffic generated in *Proactive pulling* mode. Extra traffic in *Proactive Pulling* mode is incurred by the packets requesting states from the switches. In more common cases, *Triggered Report* mode will be implemented and incurs

much less state traffic, which can be acceptable in large scale networks.

V. Related Work

Some research efforts have recently been devoted to adding state into OpenFlow data plane. Bianchi et al. [8] proposed a new abstraction to formally describe a desired stateful processing of flows inside SDN data plane based on eXtended Finite State Machines (XFSM). Moshref et al. [14] proposed FAST (Flow-level State Transitions) as a new switch primitive for SDN. Zhu et al. [22] propose a stateful abstraction for SDN data plane. All above works propose to add states to SDN. However, they mainly focus on flow-level states while ignoring equally general and complex network-level states. We propose NeSMA to support network-level state-aware applications to enrich application categories supported by OpenFlow.

Many advanced applications concerning network-level states have been proposed for better network performance and management [7], [11], [15], [18], [20]. Zhang et al. [20] propose a fast reroute mechanism to calculate the backup next hop of each prefix and directly re-route the packet when link failure happens. Animesh Patcha et al. [15] summarize current network anomaly detection mechanisms that all gather network-level states and exert statistical or machine learning processing. However, these works are constrained to support some specific states, while NeSMA provides a more general and effective mechanism to support network-level states based on the SDN central control to collect and deliver states to applications.

Some protocols for switch management including NETCONF [10], OVSDB [16] and OF-Config can monitor and configure switches during runtime. However, they could not monitor some resource and performance states of switches, and still suffer from untimely response caused by controller initiated control actions. The OpenFlow 1.5 specification proposed triggered report mode in Appendix B. 18.4. However, this field is restricted to flow level counters and cannot be configured to support network level states. To the best of our knowledge, our work is the first one on enabling network-level state-aware applications in SDN architecture.

VI. Conclusion

In this paper, we have proposed a new architecture NeSMA for building network-level state-aware applications in SDN. We have presented the detailed design of NeSMA including control and data plane extensions and APIs. We have implemented a prototype of NeSMA on Floodlight and Open vSwitch. Our experiment results shows that NeSMA can support network-level state-aware applications with low network resource consumption and high scalability, and could maintain forwarding performance. In the future, we will explore more fine-grained states in NeSMA and leverage those states to support more advanced state-aware applications. We will also investigate the possibility of hardware-based implementation of NeSMA. Besides, different state retrieval may exert different impact on switch resource occupation. Therefore, we will further evaluate the influence of each state type retrieval on forwarding performance.

VII. Acknowledgement

This research is supported by the National Natural Science Foundation of China (No.61472213 and No.61502267). Jun Bi is the corresponding author.

References

[1] Header space library: https://bitbucket.org/peymank/hassel-public.
[2] Ixia: http://www.ixiacom.cn/.
[3] Mininet: http://mininet.org/.
[4] Open vswitch: http://openvswitch.org/.
[5] Project floodlight: http://www.projectfloodlight.org/floodlight/.
[6] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.
[7] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.
[8] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.
[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 conference (SIGCOMM'13)*. ACM, 2013.
[10] Rob Enns. Netconf configuration protocol. 2006.
[11] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 27–38. ACM, 2014.
[12] Vimalkumar Jeyakumar, Mohammad Alizadeh, Changhoon Kim, and David Mazieres. Tiny packet programs for low-latency network control and monitoring. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (HotNets'13)*. ACM, 2013.
[13] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
[14] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, 2014.
[15] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer networks*, 51(12):3448–3470, 2007.
[16] B Pfaff and B Davie. The open v protocol, draft-pfaff-ovsdb-proto-01, 2013.
[17] Fabio Soldo and Ahmed Metwally. Traffic anomaly detection based on the ip size distribution. *Proceedings - IEEE INFOCOM*, 131(5):2005 – 2013, 2012.
[18] Fabio Soldo and Ahmed Metwally. Traffic anomaly detection based on the ip size distribution. In *INFOCOM, 2012 Proceedings IEEE*, pages 2005–2013. IEEE, 2012.
[19] CHANDOLA Varun, BANERJEE Arindam, and KUMAR Vipin. Anomaly detection: A survey. *Acm Computing Surveys*, 2009.
[20] Baobao Zhang, Jianping Wu, and Jun Bi. Rpfp: Ip fast reroute with providing complete protection and without using tunnels. In *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*, pages 1–10. IEEE, 2013.
[21] Shuyong Zhu, Jun Bi, and Chen Sun. Sfa: Stateful forwarding abstraction in sdn data plane. *USENIX/Open Networking Summit Research Track (ONS'2014)*.
[22] Shuyong Zhu, Jun Bi, Chen Sun, Chenhui Wu, and Hongxin Hu. Sdpa: Enhancing stateful forwarding for software-defined networking. In *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*, pages 323–333. IEEE, 2015.