Journal of
Information Technology & Software Engineering

Research Article Open Access

Achieving Memory-saving Network Update

Jiyang Liu, Liang Zhu, Weiqiang Sun* and Weisheng Hu

State Key Laboratory of Advanced Optical Communication Systems and Networks, Shanghai Jiao Tong University, Shanghai 200240, China

Abstract

Software defined networking (SDN) offers opportunities to develop high-level abstractions for implementing network update, but current SDN controller platform lacks effect mechanisms for updating network configuration on the fly. There exist two main challenges for implementing network update: 1) network is a distributed system and 2) network controller can only update one network node at a time. Naïvely updating individual nodes may lead to incorrect network behaviors. Elegant solution based on two-phase update can guarantee that traffic will be processed consistently during network update, which means each packet can be routed based either on initial network configuration or target network configuration, but never a mixture of the two. Implementing consistent network update is expensive based on previous approach, and we present mechanisms for memory-saving two-phase network update. Our design addresses one major problem: how to delete the initial configuration effectively from network nodes when controller enforces the target configuration. We propose a hierarchical network metadata structure to accelerate the procedure of removing the initial configuration. Finally, we describe the results of some experiments demonstrating the effectiveness of configuration deletion and the effect of memory-saving for the network.

Keywords: Software defined networking; Network update; Network metadata structure

Introduction

Software defined networking

The Software Defined Networking architecture, based on separation of control and data plane in network elements, enables network programmability. In today's SDN solutions, controllers are able to provide open APIs through service abstractions. For instance, an application is able to invoke connectivity services across multiple domains through a single controller. It gets underlying connectivity, by deploying a series of flow table entries to physical network nodes though the controller, at the cost of switch memory resources such as ternary content-addressable memories (TCAM), which is expensive.

Network update

From network operators' perspective, network configuration need to be modified often in cases such as network topology changes, unexpected network node or link failures, transitions in network traffic and changes in network application's policy. The emergence of SDN allows transition of networks from initial state to target state by invoking the northbound APIs of SDN platform, the control-plane then issues a sequence of Open Flow commands to deploy the target configuration to networks. The SDN platform enables a centralized view of networks and forthright manner to perform network update for network operators.

Configuration changes of network update are a common source of instability in networks, it is error prone during a network update even the initial and target configurations are correct [1]. Networks are complex systems with many distributed network nodes, network operators cannot change the whole networks in a flash because it is not possible to modify numbers of network nodes at the very same time. The total time for network nodes to accept network forwarding rules differs. Hence, to perform a network update, network operators need to do a sequence of intermediate modifications on network nodes. Owing such sequence, it generates intermediate states for networks and may exhibit unexpected behaviors that would not arise in the initial and target configurations [2].

To address these problems, researchers have proposed a graceful update mechanism called two-phase update, which guarantees that each packet be routed based either on initial or target configuration. More specifically, the idea is to pre-install the target configuration on the internal nodes, leaving the old version in place. Then, on ingress nodes, the controller sets idle timeouts, based on the Open Flow protocol, on the rules for the initial configuration and installs the target configuration at lower priority. When all flows matching a given rule finish, the rule automatically expires and the rules for the target configuration take effect [1]. However, to achieve such mechanism is expensive, it maintains forwarding rules on network nodes for both the initial and target network configurations simultaneously, and twice network memory resources consumption, during the period till the initial configuration has reached its timeout. Moreover, when multiple flows match the same rule, the rule may be artificially kept alive even though the "old" flows have all completed. If the old rules are too coarse in terms of the definition of their matching fields, then they may never die and keep consuming memory resources of the data-plane. In fact, there lack compact mechanisms to fully support efficient network update, especially in the SDN control-plane.

Our approach

From our perspective, we believe that, to achieve memory-saving network update, there need compact mechanisms to perform initial configurations deletion when networks have reached its target configurations. In order to delete the network configuration, it is necessary to locate the metadata such as network nodes and relevant flow table entries of the configuration on the control-plane. Only the relevant metadata on the control-plane get modified can the physical data-plane actually get manipulated [3,4]. Owing these, we have proposed an ideal hierarchical structure to organize metadata of configurations on the control-plane, which can accelerate the procedure of configuration retrieve and thus help to perform configuration

***Corresponding author:** Weiqiang Sun, State Key Laboratory of Advanced Optical Communication Systems and Networks, Shanghai Jiao Tong University, Shanghai 200240, China, Tel: +86 21 5474 0000; E-mail: sunwq@sjtu.edu.cn

Received December 17, 2015; **Accepted** December 22, 2015; **Published** January 09, 2016

Citation: Liu J, Zhu L, Sun W, Hu W (2016) Achieving Memory-saving Network Update. J Inform Tech Softw Eng 6: 167. doi:10.4172/2165-7866.1000167

deletion, which achieves memory-saving network update.

Related Work

Earlier research studied many distributed routing protocols [5-8] to minimize network disruptions during network routing changes. Kazemian et al. [9] proposed a graceful network abstraction model, and Reitblatt et al. [1] extended that model to well formalize network update problem. Research works in [1] introduce the notion of consistent network updates. For a given packet, the network forwarding path that a packet takes through networks, can only be in accordance with the initial configuration or the target configuration, but not a mixture of the two. It is not allowed that a packet be routed according to any intermediate state of networks during a consistent network update. And paper [1] also describes the Open Flow-compatible implementation for consistent network updates, which is called two-phase update. The basic idea of two-phase update is to explicitly tag packets upon perimeter switches, according to the version of configurations, and apply these tags as the matching fields of routing tables at each internal hop of the forwarding paths.

In the “Frenetic” project proposed in [10], two-phase update mechanisms are used to preserve consistency when performing network updates.

Based on the observation that sometimes consistency of network update can be achieved by choosing a correct order of switch updates, McClurg et al. [2] worked on ordering updates to preserve consistency during network updates. In the research, they proposed algorithms and optimizations to identify a sequence of commands to transition the network between different configurations without violating consistency invariants. Their work introduces “wait” operation for network control, to ensure that all in-flight packets of the forwarding paths based on old network configurations have left the network while the network reaches its new configurations.

Jin et al. [11] proposed a way to perform congestion-free network update. Instead of selecting one correct rule order to update networks, the paper [11] describes a new approach to dynamically select a rule order based on the update speeds of switches. They use a dependency graph to represent multiple valid orderings of rule updates, and propose algorithms to schedule updates based on the dependency graph.

The two-phase update is not always practical because that, during the transition, it consumes twice amount of switch memory resources, and it is heavy-weight. We help to eliminate the hard time for two-phase update, and we have proposed a well-defined hierarchical structure to record the relevant metadata of networks, which enables efficient transition between old and new network configurations.

This paper builds on our earlier conference paper [2], which did not include the discussion of network update. In our previous work, we proposed a network metadata structure which helps to retrieve forwarding behavior efficiently for individual network applications. It works in cases such as transitions in network traffic, changes in network application’s policy, and unexpected network node or link failures. In the current paper, we move one step further by introducing the notion of network metadata into the network update problem in general.

Network Forwarding Model

To facilitate precise reasoning about network update and state our motivation in ways of formalization, this section describes the model of network forwarding behavior and the proposed metadata structure.

The network *G* consists of a set of nodes *V* and a set of links *E*. From the perspective of SDN, the switching function behaves as follows: (1) the first packet pk_1 of flow *f* is sent by the ingress node to the controller, (2) the forwarding path *fp* for flow *f* is computed by the controller, (3) the controller installs the appropriate flow table entries at each node along the forwarding path, and (4) all subsequent packets$\{pk_2, pk_3, \ldots, pk_n\}$ in flow *f*or even different flows with matching attributes are forwarded in the data plane along the path and do not need any control plane action.

Definition 1 (Network forwarding path) As we can see from the above switching procedure of SDN, given a network flow *f* of some application, the network forwarding path fp_f is the designed switching tunnel chosen from the network topology by the controller and it consists of a set of nodes and a set of links. Formally,

$$fp_f = \{(ipt_{in}, N_{in}, ept_{in})^{tag}, \ldots, {}^{tag}(ipt_{out}, N_{out}, ept_{out})\} \quad (1)$$

where N_{in} and N_{out} are the ingress node andthe egress node of the path respectively. When a packet is forwarded across a network node, *ipt* and *ept* repesent the ingress port and the egress port of the node to forward the packet. Three-tuple array such as $(ipt_{in}, N_{in}, ept_{in})^{tag}$ is a flow table entryfor incoming flows on a network node. The *tag* label on the right side means the operation of pushing a tag on the headfield of incoming packets, while the *tag* label on the left side means performing packet matching based on the tag on packet’s head. Anorderedset of such three-tuple arraysfor a network flow forms the forwarding path.

The switching funtion of SDN behaves in ways much like the tunnel-based routing mechanisms such as MPLS. Forwarding pathsare settledin advanceforflows to be routed. Fixed network topology has limited number of forwarding elements, different forwarding paths may overlap. We thus introduce the notion of network forwarding route to represent physical forwarding tunnel for flows routed on the same tunnel.

Definition 2 (Network forwarding route) A flow traverses networks from ingress node to egress node, passing through a set of internal nodes. We organize the ordered set of nodes through which the flow passes, and define such ordered set as the forwarding route *fr*. Formally,

$$fr = \{N_{in}, N_1, N_2, \ldots, N_{out}\} \quad (2)$$

As we can see from the definition, two or more forwarding paths may be placed on the same forwarding route. The forwarding route is actually a physical processing pipeline in the network.

Definition 3 (Consistent network update) In a network update, the switching function of networks is updated with new rules. The controller changes the forwarding rules by issuing commands to nodes. Intuitively, once the forwarding behaviors of network nodes have been changed, the network is actually updated. The forwarding path is the granularity of updates in practice. The controller pushes forwarding rules to network nodes to form forwarding paths and changes forwarding paths by issuing commands according to Open Flow protocol, which we write as $fp \xrightarrow{cmds} fp'$. From the controller’s perspective, the network state *S* is the whole set of all forwarding paths. Formally,

$$S = \{fp_i \mid i = 1, \ldots, k\} \quad (3)$$

$$S' = \{fp'_i \mid i = 1, \ldots, k\} \cup \{fp''_i \mid i = k+1, \ldots\} \quad (4)$$

and we write $S \xrightarrow{u} S'$ to represent a network update *u*. Some *fp’* in *S* may be null according to thepath deletion command, and *fp’* in *S’* represents a newly added path. We call *S* stable if all nodes along with the forwarding paths are installed with new forwarding rules. And both the initial state *S* and the target state *S’* are stable. A consistent network update regulates that flows can only be processed according to forwarding paths either in the initial state S or in the target state S'during a network update.

Definition 4 (Two-phase update) In terms of the two-phase update, in the first phase, it populates the nodes in the middle of the forwarding paths with new configurations which have the tags, indicating the version of new configurations, in the matching fields. In the second phase, it enables the new configurations by installing rules at the ingress of networks and performs additional action that explicitly stamps packets with tags according to the version of configurations. Formally, suppose the initial forwarding path is

$$fp_{initial} = \{(ipt_{in}, N_{in}, ept_{in})^{tag}, {}^{tag}(ipt_1, N_1, ept_1), ..., {}^{tag}(ipt_{out}, N_{out}, ept_{out})\} \quad (5)$$

On the first phase, the forwarding path has been changed as

$$fp_{phase1} = \{(ipt_{in}, N_{in}, ept_{in})^{tag}, {}^{tag'}(ipt_1', N_1', ept_1'), ..., {}^{tag'}(ipt_{out}', N_{out}, ept_{out})\} \quad (6)$$

On the second phase, it goes like

$$fp_{phase2} = \{(ipt_{in}, N_{in}, ept_{in}')^{tag'}, {}^{tag'}(ipt_1', N_1', ept_1'), ..., {}^{tag'}(ipt_{out}', N_{out}, ept_{out})\} \quad (7)$$

During the first phase, the intermediate nodes may have been changed. The matching tag of flow table entries onboth the intermediate nodes and the egress node may have been modified. The ingress port of the egress node for coming packets may hasbeenshifted as well. When it comes to the second phase, The controller modifies boththe egress port of the ingress node for packets and the tags to push to packets' head.

To release networks from the aforementioned memory pressure during two-phase update, we perform fast process of forwarding path retrieve and deletion during the transition. We think thecontroller should remove the initialnetworkconfigurations in $fp_{initial}$ imediately after rules in fp_{phase2} have been installed on network nodes during a consistent network update.

Hierarchical Network Metadata Structure

To delete the initial network configurations after networks have deployed the target configurations, the relevant network metadata need to be located. For example, in the topology shown in Figure 1, a flow of video on demand (VOD) [12] type requests connection from node 1 to node 4. Suppose the controller deploys a forwarding path as shown below,

$$fp_{VOD} = \{(ipt_1, N_1, ept_1)^{tag}, {}^{tag}(ipt_2, N_2, ept_2), {}^{tag}(ipt_4, N_4, ept_4)\} \quad (8)$$

for the VOD flow initially. Then, after a peroid of time, the controller performs network update based on policy change, and the new forwarding path of the VOD flow is

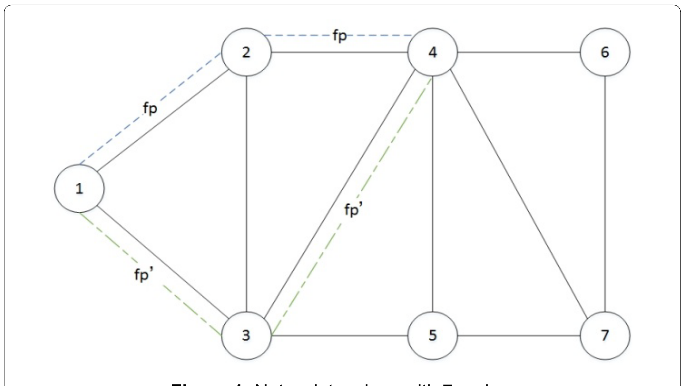

Figure 1: Network topology with 7 nodes.

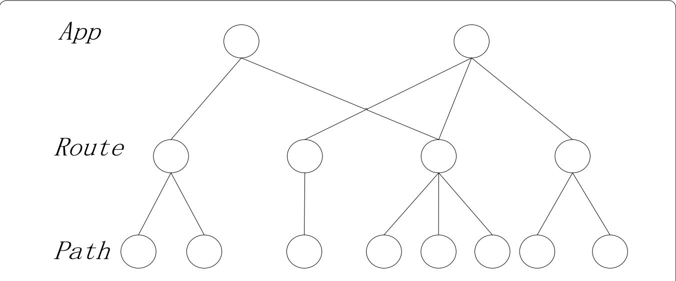

Figure 2: Tripartite graph of network forwarding metadata.

$$fp_{VOD}' = \{(ipt_1, N_1, ept_1)^{tag'}, {}^{tag'}(ipt_3, N_3, ept_3), {}^{tag'}(ipt_4, N_4, ept_4)\} \quad (9)$$

Ifthe controller wants to delete the configuration in fp_{VOD}, it first need to locate the target nodes and the corresponding flow table entries that should be removed from the network. So it is necessary for the controller to recored the information of fp_{VOD} which enables locatingthe target network metadata. In both wide area and data center networks, which deployed with many applications, the amount of forwarding paths for network flows may be tremendous large and the compact structure to organize the information of this many paths in the control-plane is a major challenge for configuration deletion.

We have proposed a structure to record network metadata such as forwarding paths and forwarding routes. In the SDN architecture, APIs are provided on the north bound, applications such as VOD can deploy policies such as bandwidth guarantee to the network control-plane through the north bound APIs. When a VOD flow requests routing through networks, the control-plane computes a forwarding path for the flow, according to the bandwidth guarantee policy as well as network conditions, and installs the forwarding path to the data-plane. The flow of VOD application then travels from an ingress node to an egress node along with the forwarding path settled by the control-plane. On account of that every forwarding path serves for its flows and each flow belongs to its network application, we introduce one more metadata of network forwarding, i.e. the network application (Figure 2).

To retrieve and remove forwarding paths efficiently, we organize forwarding paths in a hierarchical way. We abstract one potential common attribute for different forwarding paths, which is the forwarding route. Different forwarding paths may have the same physical nodes pipeline. In other words, these paths may be placed on the same forwarding route. There exists inner relation among network application, forwarding route and forwarding path.

We denote the relationship among application, route and path by tripartite graph G= ($V^{(1)}, V^{(2)}, V^{(3)}$;E), as shown in Figure 2. We define $V^{(1)}$, $V^{(2)}$ and $V^{(3)}$ and as three partition sets of G.

$$V^{(1)} = \{app_1, ..., app_l\} \quad (10)$$

$$V^{(2)} = \{route_1, ..., route_m\} \quad (11)$$

$$V^{(3)} = \{path_1, ..., path_n\} \quad (12)$$

Vertices in $V^{(1)}$, $V^{(2)}$ and $V^{(3)}$ represent applications, routes and paths respectively. There is no adjacent vertices between $V^{(1)}$ and $V^{(3)}$. Vertices in $V^{(2)}$ may have more than one adjacent vertices in $V^{(1)}$, while vertices in $V^{(3)}$ can only get one adjacent vertices in $V^{(2)}$. One vertices in $V^{(1)}$ may have multiple adjacent vertices in $V^{(2)}$. Two or more vertices in $V^{(1)}$ can share the same one adjacent vertices in $V^{(2)}$ because that one route can be shared with multiple applications. Vertices in $V^{(2)}$ cannot have multiple adjacent vertices in $V^{(3)}$ because the path is the connection which only locate in one specific route.

Once the network trigges a network update, after the target forwarding path get computed and deployed to the network, given the app tag of the new forwarding path, we can provide our first level of reference to the intial forwarding path, which is the app type of the forwarding path. Based on the reality that the initial and the target forwarding paths share the same ingress and egress nodes, given the ingress and egress nodes of the new computed path, we can provide our second level reference to the intial forwarding path to locate the route on the structure. At the third level, we get a small set of our candidate paths based on the first and second level references, and we can look upthe initial forwarding path based on the matching field on the ingress node, knowing that the intial and the target forwarding paths have the same matching field on their ingress nodes. After we retrievethe initial forwarding path, the controller can perform a sequence of Open Flow commands to delete the corresponding flow table entries on the nodes inthe data-planeaccording to the information of the initial forwarding path. The above mechanism is listed in Figure 3.

The hierarchical structure of network metadata accelerates the retrieve procedure for a specific forwarding path. Based on the multi level of references, we can locate the candidate set of forwarding paths quickly. And the amount of the candidatesis reduced enormously. After the procedure of multi level reference, we only need to check the matching field of the candidates to retrieve the desired initial forwarding path.

From our study, Onix [13] and PANE [14] matain the network metadata as the network information base (NIB), which is a database storing hosts, switches, ports queues, forwarding tables, links and their capabilities. NIB is responsble for holding network information and translating logcial actions of control to physical configurations. Modern SDN platform, such as Open Daylight [15], uses the concept of NIB to build its service abstration layer (SAL), which is the backbone of its architecture. We argue that the forwarding path of different network applications should be maintained as a meta element in NIB. Network update, which is one of the major events of SDN environment, deals with the forwarding paths. When network condition or applicationstate changes, the actionsfor forwaring paths need to be translated to network configurations. And a hirarchical way to maintain forwarding path would actually accerlerate network update operation and reduce overhead.

Implementation and Evaluation

In order to investigate the performance of forwarding path retrieve and the effect of reducing network memory overhead, both real network testbed experiments and simulations have been conducted.

Testbed experiments

We have built a prototype implementation of the hirarchical metadata structure. Open Day light is used as the controller software and we implement the proposed hierarchical structure as a plug in of Open Day light. The plugin maintains data of the structure and provides APIs to manipulate. The Mini net [2] environment is used to build the test bed network topology. A HP ProLiant ML350 Gen9 server is used as the controller host and is also connected to a set of DELL Inspiron 560 pcs running Mininet. We have designed two test cases. In both cases, a client connection reaching one of the pre-installed application servers is built randomly. We retrieve all the flow table entries of the corresponding forwarding paths in situations with and without the plugin respectively. The corresponding time cost is recorded.

In case 1, we build a test network topology with 100 network devices initially. There are 20 network applications deployed in the network. Each network node may have flow table entries related to the application's flow. We differentiate the application information of flow

```
**Procedure:** ConfigurationDelete ( $fp_{target}$ , $T$ )
**Input:**forwarding path $fp_{target}$ , hierarchical metadatatree structure $T$
**Output:**Remove $fp_{initial}$ from the network and return $fp_{initial}$
1: $app = getApp(fp_{target})$ //get the application type of the flow on $fp_{target}$
2: $src = getIngress(fp_{target})$ //get the ingress node of $fp_{target}$
3: $dst = getEgress(fp_{target})$ //get the egress node of $fp_{target}$
4: $mf_{target} = getMatchField(fp_{target})$ //get the matching field on the ingress node of $fp_{target}$
5: $T_{app} = getSubtree(T, app)$ //return a sub tree with the root node of $app$
6: $Set_{route} = getLayer(T_{app}, 2)$ //get all the elements on the second layer of the sub tree
7: **for all** $r \in Set_{route}$ **do**
8:      $src_r = getIngress(r)$
9:      $dst_r = getEgress(r)$
10:     **if** $src = src_r$ **and** $dst = dst_r$ **then**
11:         $T_r = getLayer(T_{app}, r)$
12:         $Set_{fp} = getLayer(T_r, 2)$ //get all the forwarding paths on the route $r$
13:         **for all** $fp \in Set_{fp}$ **do**
14:             $mf = getMatchField(fp)$
15:             **if** $mf = mf_{target}$ **then**
16:                 $fp_{initial} = fp$
17:                 $delete(fp_{initial})$ //remove $fp_{initial}$ from the data-plane
18:                 **return** $fp_{initial}$
```

Figure 3: Configuration delete procedure.

table entries by their destination IP in the matching fields on the ingress nodes. We initially assign 10 nodes along the forwarding path carrying the randomly built connection. Consequently, there are 10 flow table entries constructing such a connection. Then we increase the total number of network nodes and also the number of nodes in each path, to observe the time cost under different network scales. In situation with the plugin, we locate all the flow table entries in a forwarding path by iterating the structure from the top layer to the bottom layer. In situation without the plugin, we have to look up all the network nodes to find the desired flow table entries. As shown in Figure 4, situations with and without the plugin differ more than one order of magnitude in time cost.

In case 2, 20 applications are deployed in the network. We change the number of network nodes both in a path and in the whole test network to observe the time cost. The increasing number of network nodes in a path would bring more flow table entries on the path. It takes more time to retrieve a specific path consequently. On the other hand, in situation without the plugin, the time cost is affected mostly by the network scale, as listed in Figure 5. We traverse all the network nodes to find the desired flow table entries in such situation. Therefore,

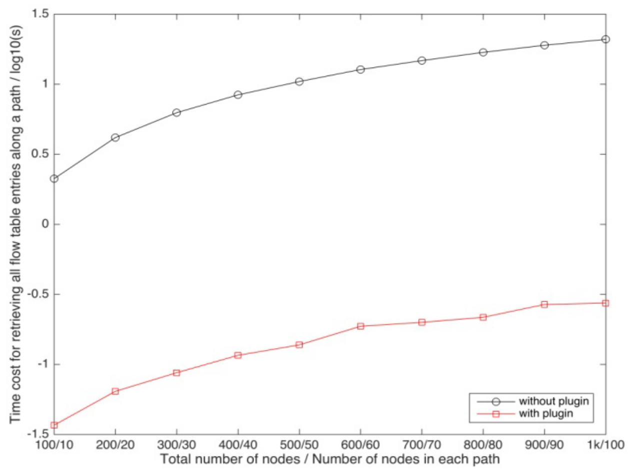

Figure 4: Time cost of retrieving flow table entries under different network scale.

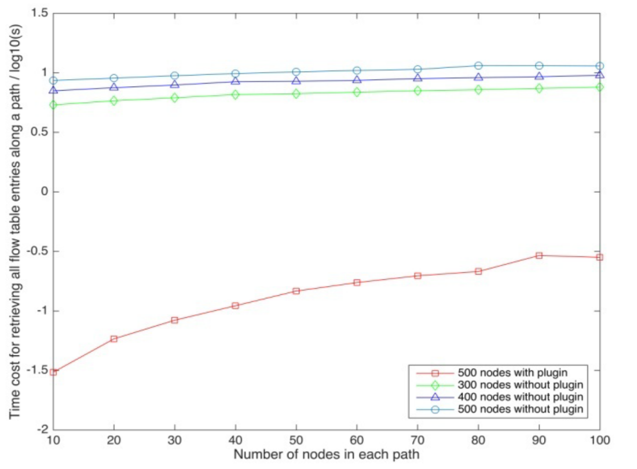

Figure 5: Time cost of retrieving flow table entries for fixed network scale with changing scale of path.

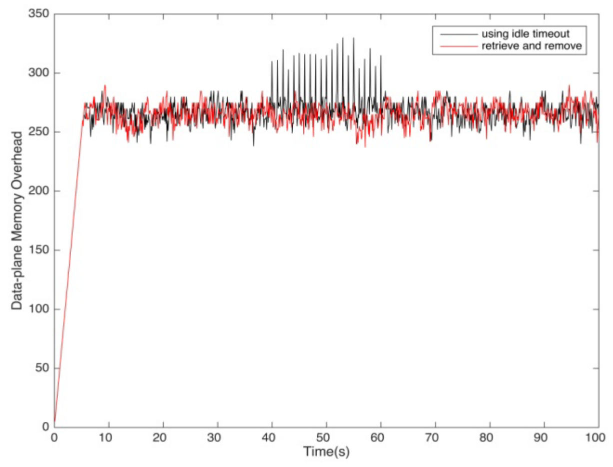

Figure 6: Data-plane memory overhead with 10 paths updated in a network update cycle.

it takes more time to locate flow table entries of an application in a larger scale of network.

Simulations

In the simulation, we use random substrate topology. According to investigations of Albert et al. [4], given two arbitrary nodes, the number of intermediate nodes are generated by using the equation below:

$$l_{rand} \sim \frac{\ln(N)}{\ln(k)} \qquad (13)$$

In Eq. (13), parameters N and k represent the whole number of nodes in a network and the average degree of network nodes respectively. In the simulation, we set K=2.66 according to [16]. The average number of nodes in a path, connecting two randomly selected nodes, is approximately equate to the right part of Eq. (13). In the simulated network topology, it is interconnected between each node. Periodically, the simulated network receives a fresh flow per 0.1 second. The idle timeout of a flow table entry is 5 second. We observe network behavior during time period of 100 seconds under different conditions.

We firstly set the scale of substrate network as 100 nodes and the average node number of forwarding paths is five, according to Eq.(13). During the time interval between the 40th second and the 60th second, we update the simulated network at each second. In Figure 6, we update 10 forwarding paths in an update cycle and record the data-plane memory overhead at different time points. Data-plane memory overhead is understood as the total amount of flow table entries in the substrate network. It is observed that the two-phase update, using idle timeout to uninstall configurations, leads to a sharp increase of memory overhead, while our proposed mechanism to remove configurations performs a stationary update process. In Figure 7, we increase the number of forwarding paths to be transformed to 20 in an update cycle. As we can see between Figures 6 and 7, the data-plane memory overhead rises as the scale of network update increases under the condition using idle timeout. While in situation with our mechanism, the update process keeps stationary at a coarse granularity. With the increasing scale of network update, mechanism of idle timeout for two-phase update brings more and more pressure to data-plane memory. Network traffic behavior differs a lot between day and night, and at some points of time, the network need to deploy large scale update to meet traffic

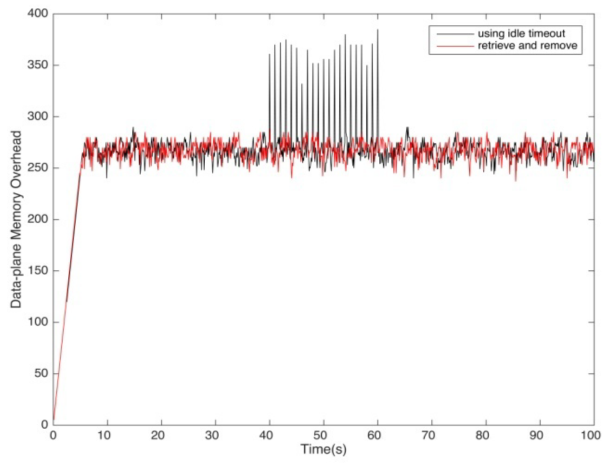

Figure 7: Data-plane overhead with 20 paths updated in a network update cycle.

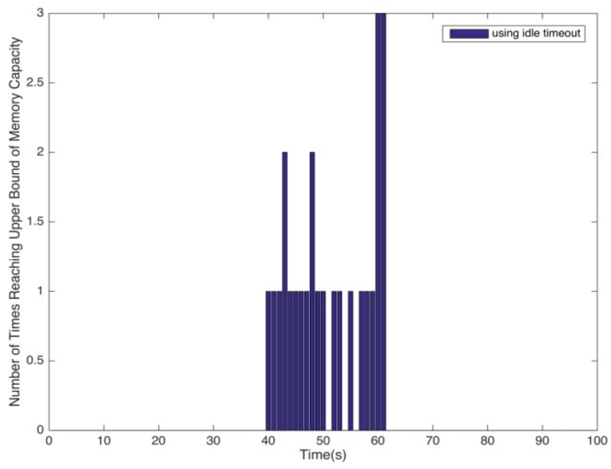

Figure 8: Number of times reaching upper bound of memory capacity of a node in two-phase update case using idle timeout.

characteristic. The two-phase update with our proposed mechanism can apply well to such scenario.

Next, we investigate the memory overhead of key nodes under a fixed topology as shown in Figure 1. There are 7 nodes totally and node 4 has the highest degree. In the simulation, since each node has the same probability to receive new flows, we think node 4 holds more traffic than others. We update the network at each second during the time period between the 40th second and the 60th second, and 10 forwarding paths are changed in every update process. We set the upper bound capacity of memory for each node to hold flow table entries as 25 and record the number of times for a specific node to reach its memory capacity in the span of one second. It is obvious from Figures 8 and 9 that our mechanism can help to ease off the memory overhead of network key node.

Now that we investigate the impact of network scales. We observe the memory overhead under different network scales with 100, 200 and 300 nodes respectively, and the corresponding number of nodes along a forwarding path is set as 5, 6 and 7. The simulation process updates the network at each second between time period from the 10th second to the 30th second for the 300 nodes network, from the 40th second to the 60th second for the 200 nodes network, and from the 70th second to the 90th second for the 100 nodes network. 20 paths, 40 paths and 60 paths are updated in an update process for network scales of 100 nodes, 200 nodes and 300 nodes, respectively. The results in Figures 10 and 11 show that the memory overhead increases as the network scale grows, during the update process using idle timeout mechanism. In contrast, with the increasing network scale, the retrieve and remove mechanism using a hierarchical structure brings little impact to the memory overhead during the network update (Table 1).

Conclusion

In this work, we are interested in the memory overheadduringthe two-phase network update process. We provide a solution based on configuration deletion. We argue that the challenge is to realize efficient configuration retrieve, and we show that a network metadata structure to organize information of forwarding path in a hirarchical way which could accelrate the retrieve process, and should be designed as a meta

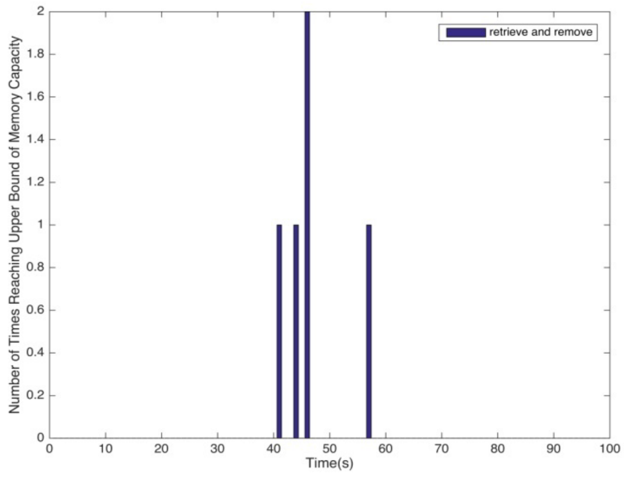

Figure 9: Number of times reaching upper bound of memory capacity of a node in two-phase update case using proposedmechanism.

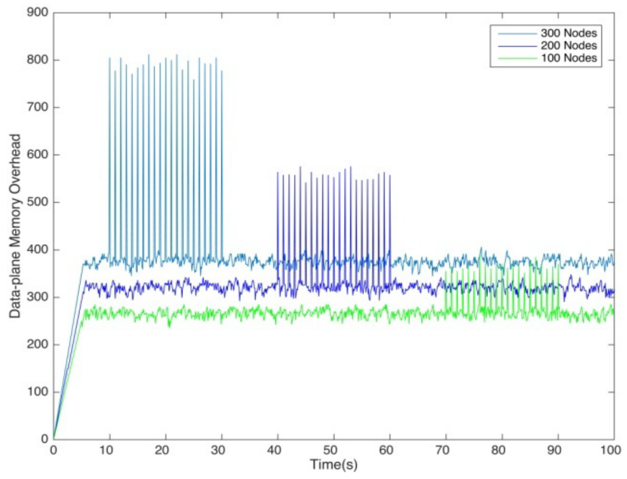

Figure 10: Data-plane memory overhead under different network scale in two-phase update case using idle timeout.

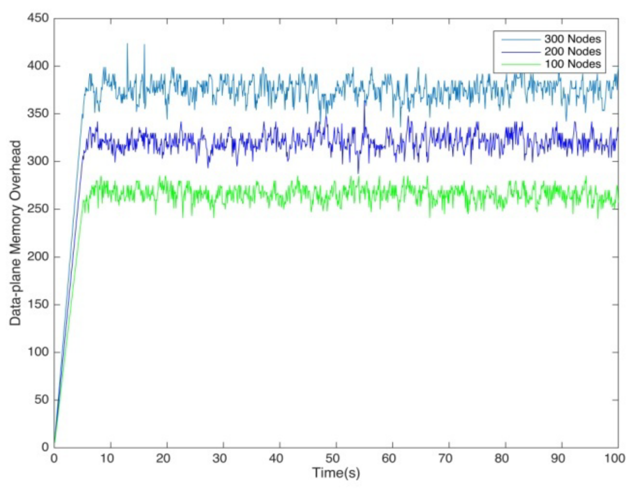

Figure 11: Data-plane memory overhead under different network scale in two-phase update case using proposed mechanism.

Symbol	Meaning
G	Physical network
V	Set of network nodes
E	Set of network links
pk	Packet
f	Flow
fp	Forwarding path
fr	Forwarding route
N	Network node
ipt	Ingress port of nodes for a packet
ept	Egress port of nodes for a packet
S	Network state

Table 1: Network forwarding Notations.

component in the network information base. We investigate the performance in both simulation and testbed environments [17]. The results show that the proposed mechanism outperforms the original one and brings performance enhancementin time cost for retrieving a forwarding path and in memory overhead under different scenarios.

Acknowledgement

This work is supported by NSFC (61433009, 61431009,61271217), and Ministry of Education (20110073130006).

References

1. Reitblatt M, Foster N, Rexford J, Schlesinger C, Walker D (2012) Abstractions for network update. In Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication. USA.
2. McClurg J, Hojjat H, Cerny P, Foster N (2014) Efficient synthesis of network updates. arXiv preprint arXiv.
3. www.mininet.org
4. Albert R, Barabási AL (2002) Statistical mechanics of complex networks. Reviews of modern physics, Indiana.
5. Liu J, Zhu L, Sun W, Hu W (2015) Scalable application-aware resource management in software defined networking. In Transparent Optical Networks (ICTON), 2015 17th International Conference on, Budapest.
6. Francois P, Shand M, Bonaventure O (2007) Disruption free topology reconfiguration in OSPF networks. In INFOCOM 2007, 26th IEEE International Conference on Computer Communications,Anchorage, AK.
7. Francois P, Bonaventure O, Decraene B, Coste P A (2007) Avoiding disruptions during maintenance operations on BGP sessions. Network and Service Management, IEEE Transactions on.
8. Raza S, Zhu Y, Chuah CN (2011) Graceful network state migrations. IEEE/ACM Transactions on Networking (TON), USA.
9. Kazemian P, Varghese G, McKeown N (2012) Header Space Analysis: Static Checking for Networks. In Networked Systems Design and Implementation (NSDI), USA.
10. Vanbever L, Vissicchio S, Pelsser C, Francois P, Bonaventure O (2011) Seamless network-wide IGP migrations. ACM SIGCOMM Computer Communication Review, USA.
11. Jin X, Liu HH, Gandhi R, Kandula S, Mahajan R, et al. (2014) Dynamic scheduling of network updates. In Proceedings of the 2014 ACM conference on SIGCOMM, USA.
12. Foster N, Guha A, Reitblatt M, Story A, Freedman MJ, et al. (2013) Languages for software-defined networks. Communications Magazine, IEEE 51: 128-134.
13. Koponen T, Casado M, Gude N, Stribling J, Poutievski L, et al. (2010) Onix: A Distributed Control Platform for Large-scale Production Networks. InOperating Systems Design and Implementation (OSDI), USA.
14. Ferguson AD, Guha A, Liang C, Fonseca R, Krishnamurthi S (2013) Participatory networking: An API for application control of SDNs. In ACM SIGCOMM Computer Communication Review, USA.
15. Jarschel M, Wamser F, Hohn T, Zinner T, Tran-Gia P (2013) Sdn-based application-aware networking on the example of youtube video streaming. In Software Defined Networks (EWSDN), 2013 Second European Workshop on, Berlin.
16. www.opendaylight.org
17. Govindan R, Tangmunarunkit H (2000) Heuristics for Internet map discovery. In INFOCOM 2000,19th Annual Joint Conference of the IEEE Computer and Communications Societies, USA.