Routing-State Abstraction Based on Declarative Equivalence

Kai Gao[†] Xin Wang[×] Jun Bi[†] Guohai Chen Andreas Voellmy[+] Y. Richard Yang[×+]
[×]Tongji University [†]Tsinghua University [+]Yale University

ABSTRACT

Providing abstract views on top of raw network state can provide substantial benefits to both the network OS, who manages the network state, and the network control applications, who consume the network state. In this paper, we conduct the first study to provide control applications with access to the routing state, which is a key component of the network state. We design a simple, efficient algorithm to look up a route query in a flow rule manager (FRM), which is a common data structure storing a network's routing state. More importantly, we design a simple, novel interface based on a principle called *declarative equivalence* for network control applications to query the routing state, and the network OS uses redundancy elimination to compute equivalent, but minimal routing state, providing the first, novel, systematic algorithm to compute abstract, compressed routing state. We implement our design in OpenDaylight and show substantial performance benefits.

1. INTRODUCTION

A major lacking of the current SDN architecture is the missing of key abstractions in the control plane, where distribution, configuration, and forwarding are identified as the three key components of SDN [11]. Specifically, distribution collects a global network state to allow centralized control programming; configuration provides abstract views of the centralized state to simplify querying (read) and/or programming (write) network state; and forwarding provides a unified model of the distributed forwarding elements. In the last few years, substantial progress has been made in distribution (*e.g.*, the design and implementation of OpenDayLight [10] and ONOS [8]) and forwarding (*e.g.*, OpenFlow 1.0 [7] to 1.4 [9] and P4 [3]). However, little progress has been made on configuration abstractions.

Providing abstract views on top of raw network state can provide substantial benefits to both the network OS, which manages the network state, and the network control applications, which consume the network state. First, a more compact abstract network state view can reduce the requirement on client scaling. The raw network state of a large network may consist of a large number of network devices. A consumer of such a large amount of information must be scalable. Second, an abstract network state view can better protect the privacy of the provider of the network. Third, an abstract network state view may substantially reduce the load of information updates.

Despite the aforementioned substantial benefits, there are few systematic studies on constructing abstract views on top of raw network state. The existing abstract network-state views, such as the single-node view (*e.g.*, NOX [5], ALTO [1]), are statically constructed, simple views. In the general case, let $view()$ be the function that constructs an abstract view for a network control function f. One can see that $view()$ may compute an on-demand, instead of static view that will depend on not only f but also functions $\{f'\}$, which construct the network state that f depends on. For example, let f be a flow-rate scheduling function, for fixed routing. Then $\{f'\}$ will be the functions that contribute to the construction of the routing in the network. The ultimate goal of our project is to develop a systematic framework to design $view()$.

In this short paper, we make a first step toward realizing the preceding goal. In particular, we focus on control applications who need to access the routing state of the network OS. We say that these applications request the routing-state query service of the network OS. As routing is the most basic service of a network, the routing-state query service as one of the most basic services of network state abstraction.

Providing an efficient, compact routing-state query service, however, is not trivial. First, as a basic step, consider a control application who queries for the route (or the properties of the route) of a flow (*e.g.*, a Web session). Looking up the route in the data structure of the network OS storing the routing state, however, may not be trivial. In particular, current network OS'es such as OpenDayLight and ONOS use a data structure called flow rule manager (FRM) to aggregate the effects of potentially complex routing computation involving many components (*e.g.*, customized routing). As we will show, looking up the route in FRM of a customized route query is not trivial; there is no existing published algorithm to address this issue.

Further, and more importantly, it may not be desirable at all to return raw routes in providing the routing-state query service, due to privacy, scalability concerns, as we already discussed in the general setting. For example, a flow-rate scheduling application (assume route is given) may not need to know the exact routes of all source-destination pairs that the application schedules, as long as shared bottlenecks are given. Revealing minimal route information is a key benefit of routing-state abstraction, but there is no existing algorithm that computes abstract, minimal routing states.

The key contribution of this paper is that we conduct the first study to address the preceding two issues. In particular, we design a simple, efficient algorithm to look up a route query in FRM. More importantly, we design a simple, declarative API for a network control application to specify its need (*i.e.*, requirements) of routing and topology state,

and the network OS, using redundancy elimination as a key step, computes the minimal, but equivalent routing state, providing the first, novel, systematic algorithm to compute abstract routing states. We implement our design in OpenDaylight and evaluate its performance. Even for a small topology, we achieve compression ratio close to 5.

We emphasize that our design is still in a relatively early stage. It provides a foundation for potential standard [2], and opens up substantial new capabilities that should be explored but we have not explored in this paper. In particular, the completeness of our design is a major future work item, although we provide a basic completeness result.

The rest of this paper is organized as follows. Section 2 gives examples to illustrate both the use cases and the issues to be solved to support the use cases. In Section 3, we give an overview of design. In Section 4, we give the novel algorithms to compute abstract routing state based on redundancy elimination. Section 5 presents evaluation results.

2. MOTIVATION

We start with examples to both provide basic use cases of the routing-state query service and illustrate the key issues to be solved. We choose the examples to be as simple as possible for illustration purpose. Real life examples are more complex and we evaluate some in Section 5.

Figure 1 is the example network, which has 7 switches ($\texttt{sw}_1$ to $\texttt{sw}_7$). Switches $\texttt{sw}_1$/$\texttt{sw}_3$ provide access on one side, $\texttt{sw}_2$/$\texttt{sw}_4$ provide access on the other side, and $\texttt{sw5} - \texttt{sw7}$ form the backbone. End-hosts (eh1 to eh4) are connected to access switches sw1 to sw4 respectively. Assume that the bandwidth of each link is 100 Mbps.

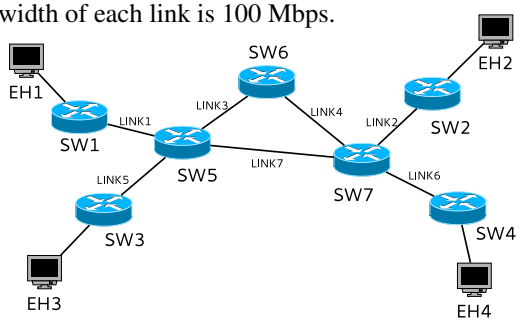

Figure 1: Example network.

With the flexibility of new control capabilities (*e.g.*, SDN), the network uses a customized routing algorithm. Furthermore, assume that the network implements access control using Group Based Policy (GBP) [4] as follows.TODO. Assume that the network uses an active SDN controller such as OpenDayLight or ONOS, the final effects of the preceding algorithms will be aggregated into the flow rule store managed by the flow rule manager.

Basic routing state query: First, consider the use case that the administrator of the network is asked about the total available bandwidth of an FTP session from eh1 to eh2. To answer this question, the administrator should be able to issue a query command to determine the route from eh1 to eh2.

The storage of the routing state in FRM, however, does not provide a direct answer to the preceding query. The administrator could display the whole FRM and try to manually match the rules in each switch to determine the route, but this is not productive.

Routing state abstraction: Now, consider a higher-layer network control application, which conducts rate control. Assume that the routing state query is to obtain the available bandwidth of two flows: eh1 to eh2, and eh3 to eh4, respectively. Simply returning a number for each flow may not be sufficient. Assume that the returned result is 100 Mbps for each flow. But the application cannot determine that if it schedules the two flows together, whether it will obtain a total of 100 Mbps or 200 Mbps. This depends on whether the routing of the two flows shares a bottleneck in the current routing state:

1. If the two flows use different paths, for example, when the first uses sw1 $\rightarrow$ sw5 $\rightarrow$ sw7 $\rightarrow$ sw2, and the second uses sw3 $\rightarrow$ sw5 $\rightarrow$ sw6 $\rightarrow$ sw7 $\rightarrow$ sw4. Then the application will obtain 200 Mbps.

2. On the other hand, if the two flows share a bottleneck, for example, when both use the direct link sw5 $\rightarrow$ sw7, then the application will obtain only 100 Mbps.

To allow applications to distinguish the two possible cases, the network needs to provide more details on the routing state. A naive solution to this problem, then, is to return the two complete, detailed routes and the available bandwidth of each link on the routes. But this may not be desirable, as the application may not need the details and/or may not have the permission to see networks details.

Now consider what route abstraction can achieve. Assuming case 2 (shared bottleneck), it is sufficient for the net OS to return a single abstract link for each flow: $\texttt{ane}_1(\leq \texttt{100Mbps})$, where ane stands for abstract network element.

Consider a variation of the preceding case. Assume that the capacity of the link from $\texttt{sw}_1$ to $\texttt{sw}_5$ is 70 Mbps, while the rest are still at 100 Mbps. Then the abstract route from eh2 to eh4 becomes $\texttt{ane}_1(\leq \texttt{100Mbps})$ and $\texttt{ane}_2(\leq \texttt{70Mbps})$.

For simple networks such as the example, one can manually construct the minimal, abstract routes as above. For real networks with complex topologies and many flows, an efficient algorithm is essential, and this is a key problem that we solve in this paper.

3. SERVICE OVERVIEW

We now present our design. This section gives the API, the basic work flow, and then the basic routing-state lookup algorithm. The details of the route abstraction/compression algorithm is specified in the next section.

3.1 The API and Basic Work Flow

The more the network OS knows about what a network control application f needs regarding a routing state query, the more concise the network OS response can be. Hence, an extreme API is that the complete network control application f (*i.e.*, the code and related state) is sent to the net-

work OS. This, however, can create substantial complexity in the routing-state query component, as even some simple program properties (*e.g.*, halting) are already difficult to analyze. Also, in settings such as interdomain, the owner of the control function f may not want to provide the complete f to the network OS. Another extreme API is that each routing state query provides only the most basic information (*i.e.*, the source and the destination). This, however, does not provide enough information for the routing-state service to compute efficient route abstraction/compression. Hence, the returned routes will be independent of individual control functions, missing out opportunities on abstraction or compression.

Our API strikes a careful balance between the two extremes, using a principle that we call *declarative equivalence*. Specifically, in a declarative-equivalence design, each network-state query provides a declarative description of the control function f, to describe what it needs regarding the network state query, so that the network OS provides a simpler, abstract, but *equivalent* network state, according to the equivalence condition in the declaration. Instantiating the preceding in the context of routing-state query, Figure 2 gives the grammar to specify the API that a control application requests the routing-state query service:

```
rs-query   := flow-list equiv-cond
flow-list  := flow [flow-list]
flow       := generic-match-condition
```

Figure 2: The Routing-state query API.

Specifically, the key component of a routing-state query is a list of flows (`flow-list`). Traditional protocols such as PCEP and ALTO specify a flow by providing only a source IP address and a destination IP address. The emergence of SDN, however, provides much routing flexibility and hence the route of a packet can depend on not only the IP addresses but also other fields such as TCP/UDP ports. Thus, it is necessary that a query to the routing state also provides the same flexibility. As a result, each flow in the API is specified by a match condition, as in OpenFlow. For simplicity of discussion, we focus on unicast flows in this paper.

Another key component of a routing-state query is the declared equivalence condition. A particular type of equivalence condition, in the context of routing-state query, is the equal range condition. We give the detailed specification of the condition in Section 4.

After receiving a routing-state query, the network OS retrieves the route for each flow, and then computes the result after compression (abstraction). Our API provides an indicator to allow the control application to receive updates to the query results, achieving push notification. The push notification is implemented using HTTP SSE.

3.2 FRM Lookup

Problem: A basic step in the work flow is to lookup the route of each flow specified in a routing-state query. As we illustrate in Section 2, a key issue is due to potential mismatch of the match in the routing-state query and the current routing state. Hence, the key idea of our lookup algorithm is maintaining a union of match of available rules and skipping the rule which has a match included in the union match even it has an intersection with the queried match. 'Available' here means the rule has an intersection with the match in the routing-state query and also it isn't covered by higher priority rules. Note that the algorithm assumes that the full routing state is stored in FRM. Hence, we consider a proactive routing system. In the case of reactive routing (*e.g.*, [12]), the FRM may be only a cache of routing decisions. Since proactive routing is a more common routing design, we focus on this setting and leave the case of reactive routing as future work.

Algorithm 1 Find *ONE* Path in FRM

Require:
  R - the set of rules stored in FRM
  S - the set of switches in the network
  M - the match of flow
  l - the ingress link
Ensure: P - the matching paths
  procedure MERGERULE(*match*, *action*, r, M)
    $I \leftarrow$ *Intersection*(*Match*(r), M)
    if $I \neq \emptyset$ and $I \nsubseteq$ *match* **then**
      match $\leftarrow$ *match* $\cup$ I
      action $\leftarrow$ *action* $\cup$ *Action*(r)
  function FRMLOOKUP(R, S, M, l)
    visited $\leftarrow \emptyset$, *path* $\leftarrow \emptyset$
    $p \leftarrow$ the ingress port for l
    $s \leftarrow$ the switch that owns p
    while true **do**
      if $p \in$ *visited* **then**
        return $\emptyset$ ▷ Loop
      visited $\leftarrow$ *visited* $\cup \{p\}$
      match $\leftarrow \emptyset$, *action* $\leftarrow \emptyset$
      $M' \leftarrow M \cup \{$*ingress_port* $= p\}$
      $R_s \leftarrow$ rules on s
      while $R_s \neq \emptyset$ **do**
        $r \leftarrow$ the *rule* with highest priority in R_s
        $R_s \leftarrow R_s \setminus \{r\}$
        MergeRule(*match*, *action*, r, M')
      if $M' \neq$ *match* **then**
        action $\leftarrow$ *action* $\cup \{$*drop*$\}$
          ▷ Rules for unmatched flows
      if *action* has arbitary destinations **then**
        return $\emptyset$ ▷ We don't want arbitary paths
      if *action* is *drop* **then**
        return $\emptyset$
      $s' \leftarrow$ next-hop(*action*)
      $p \leftarrow$ the ingress port for (s, s') on s'
      path $\leftarrow$ *path* $\cup \{(s, s')\}$
      if (s, s') is an egress link **then**
        return *path*
      $s \leftarrow s'$

Algorithm 1 gives the lookup algorithm. It simulates the process of packet forwarding to find the path for a given flow

characterized by a match condition M. The algorithm starts from an ingress port and the corresponding ingress switch. For each switch it goes from the flow rule with the highest priority to the one with the lowest and computes the *intersection* of the rule's matching condition and M, denoted by I. There are two cases when a rule will be deprecated: $I = \emptyset$ which indicates that the rule has no effect on this flow, and $I \subseteq$ *match* which means this rule is overlapped by another one with a higher priority. The variable *match* represents the subset of M that is already matched by some rule. It will be updated every time a new matching rule is found and so will the *action* set.

Currently we do not support multiple paths so the algorithm would report that no valid path is available when the *action* set contains arbitary operations, such as pushing vs. no pushing *vlan* tags, routing to different ports, etc.

If the actions for M on a switch are identical, either the flow is dropped or an egress port is provided Dropping the flow is considered a sign that the path is not valid so we return an $\emptyset$. As with the egress port the algorithm adds its corresponding link to the *path*. If the paired ingress port is from an external switch, the algorithm terminates and returns the *path*. Otherwise it will proceed with this ingress port unless a loop is detected.

4. ROUTE ABSTRACTION/COMPRESSION

After retrieving the route(s) of each flow, a simplistic routing-state service will just return all links (and their attributes) on the routes. This section develops algorithms to compute abstract, compressed routing state.

4.1 Problem Definition

Ntations: Let $attr$ be a vector for a given link attribute. Let vector $R[i]$ represent the result of route lookup for flow i, where $R[i][e]$ is the fraction of traffic of flow i on link e, according to the current routing state. For example, the result of route lookup for the second use case in Section 2 can be represented as the following:

```
       R[0] R[1] avabw delay bg-tr
link1  1    0       1G   2ms  ...
link2  1    0     100M   5ms  ...
link3  1    1     100M   5ms
Link4  1    1     100M   5ms
link5  0    1     100M   7ms
link6  0    1       1G   4ms
...
linkM
```

Although a routing-state query without abstraction/compression will return all of the data shown above, route abstraction/compression will select only a subset link attributes (columns) and some links (rows). Elimination of links from the complete result achieves compression but may result in loss of information to the control application. Hence, a specification on conditions whether the elimination of a set of links from the complete result leads to information loss or not is the key to the problem definition. Such a specification, however, can be provided only by the application itself.

Insight: Specifically, in the general case, the result from the routing-state query will become the input parameters for the algorithms in the control application, to help the control application to make decisions. Let $\mathbf{x}$ be the vector of the decision variables in the control application. Then, one can identify that a generic structure of the control application is to solve/optimize $obj(\mathbf{x})$, subject to two types of constraints on $\mathbf{x}$: (1) those do not involve the results from the routing state query; and (2) those do. Let the first type limit $\mathbf{x} \in \mathbf{X_0}$. Consider the second type. The state of art in algorithmic design typically handles only linear constraints, and hence the set S of constraints of this type will be of the format $\{\mathbf{a}_k^T\mathbf{x} \leq b_k\}$, where $\mathbf{a}_k$ is a vector, and b_k a constant. Hence, it is in $\mathbf{a}_k$ or b_k where the result from the routing-state query appears. Let $\mathbf{Ax} \leq b$ as a matrix format to represent the whole set of constraints.

Now, consider the case that a link appears in the complete result of a routing-state query, but its parameters do not appear in a boundary constraint among the aforementioned constraints, then the link may not need to appear in a compressed routing-state query result.

DEFINITION 1 (EQUIVALENCE). *Two constraint sets S_1 and S_2 of a network control function are **equivalent** if and only if they limit the decision variables in the same way:* $\mathbf{X_0} \cap \{\mathbf{x} : A_1\mathbf{x} \leq \mathbf{b}_1\} = \mathbf{X_0} \cap \{\mathbf{x} : A_2\mathbf{x} \leq \mathbf{b}_2\}$

DEFINITION 2 (REDUNDANT). *A constraint s is **redundant** to a constraint set S if and only if $s \in S$ and the two sets S and $S \setminus \{s\}$ are **equivalent**.*

DEFINITION 3 (MINIMAL CONSTRAINT SET). *A constraint set S is **minimal** if and only if for any $s \in S$, s is not **redundant**.*

DEFINITION 4 (EQUIVALENT ROUTING-STATE QUERY). *A declarative equivalence based routing-state query is one where the querier (control application) declares $\mathbf{X_0}$ and a set of constraints $S = \{\mathbf{a}_k^T\mathbf{x} \leq b_k\}$. If the attribute of a link does not appear in a minimal constraint set, the link can be eliminated from the routing-state result.*

A concern one may have is that the preceding definition may be limited. Consider the case of hierarchical networks, where the upper-layer network (*i.e.*, the network control application) conducts routing (traffic engineering) in its layer and uses routing-state query to obtain the state of the lower layer (*i.e.*, the network OS). Let flows be the $n(n-1)$ source-destination pairs in the upper layer network with n nodes. Let $\mathbf{x}$ be the set of decision variables controlling the routing in the upper-layer, where each element is the routing on each of the preceding flows. Let X_0 encode the constraints on traffic demand. We have the following result:

PROPOSITION 1 (UTE COMPLETENESS). *Any upper-layer routing (traffic engineering) algorithm where the goal of routing-state query in the lower-layer network is to avoid*

congestion of shared links or shared risk groups can be implemented using the declarative equivalence based routing-state query. We refer to this as the upper-layer traffic engineering (UTE). Let $\mathbf{A} = \mathbf{R}$ *and* $\mathbf{b} = \mathbf{cap}$*. Then the routing-state query returns a link only if the link may become a bottleneck in the upper layer network.*

4.2 A Basic Algorithm

Although the preceding definition defines the problem precisely, it does not provide an efficient algorithm yet. A naive algorithm is for the network OS to try out each link, but this can lead to an exponential algorithm. We now develop practical algorithms to eliminate the redundant constraints. The algorithm is based on Lemma 2; see our technical report for proofs on the formal results in this paper.

LEMMA 1 (SUBSET PRODUCES BETTER SOLUTION). *If* $S_0 \subseteq S_1$*, for any objective function*

$$y = \max \mathbf{a}^T \mathbf{x}$$

the corresponding solutions y_0 *and* y_1 *with constraints* S_0 *and* S_1 *respectively satisfies that* $y_0 \geq y_1$*.*

LEMMA 2 (REDUNDANCY CRITERION). *When* $\mathbf{x}$ *is a vector of continuous variables,* $\forall s : \mathbf{a}_s^T \mathbf{x} \leq b_s \in S$*, s is not redundant if and only if* b_s *is less than the solution to the objective function*

$$y = \max \mathbf{a}_s^T \mathbf{x}$$

on the constraints of

$$A'\mathbf{x} \leq \mathbf{b}'$$

where A' *is the matrix form of set* $S \setminus \{s\}$*.*

In this subsection we propose Algorithm 2 and prove its correctness. The algorithm generates the **minimal equivalent** subset of a given constraint set based on Lemma 2.

Algorithm 2 Find the Minimal Equivalent Constraint Set (1)

Require: S - the set of linear constraints
Ensure: S_0 - the minimal equivalent subset of S
  function MINEQUIVCONSTRAINTSET(S)
    $S_0 \leftarrow \emptyset$
    for $s \in S$ **do**
      $s_1 \leftarrow$ transform s into standard form $\mathbf{a}_1^T \mathbf{x} \leq b_1$
      constraints $\leftarrow S \setminus \{s\}$
      obj_func $\leftarrow \mathbf{a_1}$
      $y \leftarrow$ SolveLP(*constraints*, *obj_func*)
      if $b_1 < y$ **then**
        $S_0 = S_0 \cup \{s\}$
    return S_0

THEOREM 1 (CORRECTNESS OF ALGORITHM 2). S_0 *is* ***minimal*** *and* S_0 *is* ***equivalent*** *to* S*.*

4.3 Extension: Fast Elimination Algorithm for 0-1 Constraints

The performance of Algorithm 2 can be improved when the constraints have certain properties. As in the case of upper-layer routing, in the case of single-path routing, one can observe that each entry in the matrix is either 0 or 1. This simple property, however, allows substantial speedup.

LEMMA 3. *When* $\forall s \in S$*, s can be transformed into the format of* $\mathbf{a}_s^T \mathbf{x} \leq b_s$ *and* $\mathbf{a}_s \in \{0,1\}^n \setminus \{\mathbf{0}\}, \mathbf{x} \in {R_+}^n, b_s \in R_+$*, let* S_0 *denote the* ***minimal equivalent*** *subset of* S*.*

If there is only one element s_0 *with the minimal capacity,* $s_0 \in S_0$*. If there are multiple elements with the minimal capacity, the one with most 1 coefficents must belong to* S_0*.*

From Lemma 3 we can conclude that a certain element of the constraint set is always in the **minimal equivalent** subset. Also it has specified how to find such a constraint. Utilizing the lemma, we develop Algorithm 3.

Algorithm 3 Find the Minimal Equivalent Constraint Set (2)

Require: S - the set of linear constraints
Ensure: S_0 - the minimal equivalent subset of S
  function MINEQUIVCONSTRAINTSET(S)
    $S_0 \leftarrow \emptyset$
    $S \leftarrow \{s_1 \leftarrow$ the standard form of $s, \forall s \in S\}$
    while $S \neq \emptyset$ **do**
      $s_0 \leftarrow$ *FindMinimalElement*(S)
      if $S_0 = \emptyset$ **then**
        $S_0 \leftarrow S_0 \cup \{s_0\}$
      else
        constraints $\leftarrow S_0$
        obj_func $\leftarrow$ coefficients(s_0)
        $y \leftarrow$ SolveLP(*constraints*, *obj_func*)
        if $b_{s_0} < y$ **then**
          $S_0 \leftarrow S_0 \cup \{s_0\}$
      $S \leftarrow S \setminus \{s_0\}$
    return S_0

PROPOSITION 2 (CORRECTNESS OF ALGORITHM 3). *Algorithm 3 will return the* ***minimal equivalent*** *subset of* S*.*

4.4 Summary

Both algorithms invoke *SolveLP* $|S|$ times. The amount of constraints increases from 1 to a maximum number of $|S|-1$ in Algorithm 3 while it always stays $|S|-1$ in Algorithm 2. However, Algorithm 2 has less requirements on the constraint set and thus are capable of handling generic problems. Also the performance can be enhanced using parallel computing techniques.

5. EVALUATIONS

In this section, we evaluate the benefits of routing abstraction for the UTE use case (Section 4.1) to show that our design (1) generates compressed routing states, (2) improves the performance of UTE, and (3) achieves routing states push notification efficiently.

#Flow	#Origin	#Compressed	Rate	RT3(ms)	RTG(ms)
5	19	4	0.21	375	692
10	30	7	0.23	764	1843
20	45	12	0.27	866	1636
30	56	18	0.32	1543	3621
40	62	23	0.37	2493	4094
50	67	27	0.40	3403	5975
60	67	26	0.39	3416	6408
70	72	40	0.56	5431	13955
80	68	27	0.40	3550	10077
90	73	39	0.53	6173	18643
100	69	41	0.59	6557	21696

Table 1: Compressed links, running time at different number of flows

5.1 Compressed routing states

We first evaluate the compression rate and performance of routing abstraction.

Topology: We use the topology from [6] and name it *Internode*. It has 65 switches and 78 links. We divide the switches 20 internal and 45 external switches. The links between internal switches have higher bandwidth than those between external switches. We use shortest path as base routing.

Results: Table 1 shows the results of the number of compressed links, compression rate, and running time of Algorithms 2 and 3, as we vary the different number of flows. $\#Flow$ gives the number of flows in a query. We make the following observations. First, our algorithms can achieve large compression ratios. Column 2 ($\#Origin$) shows the origin number of links and column 3 ($\#Compressed$) gives the compressed number of links. We can see that with 5 flows, our algorithm reduces the number of links by a factor of almost 5.

Second, Algorithm 3 can speedup compression. Column 5 ($RT3$) gives the running time of Algorithm 3 and column 6 (RTG) is the running time of a general compression algorithm that filters links by removing one link and computing optimum for the rest by LP. We can see that as the number of flows increasing, for example, for 100 flows, Algorithm 3 can be 3 times faster.

5.2 Push notification

We then evaluate the performance of push notification.

Update cases: We evaluate four cases of routing state change. Specifically, we divide the links into two sets: S_1 and S_2, where S_1 is the set of links that should be sent to higher-layer application and $S_2$the rest. Considering the available bandwidth of links could increase or decrease, we can divide link capacity updates into 4 cases: (1) increasing at S_1 ($case1$), (2) decreasing at S_1 ($case2$), (3) increasing at S_2 ($case3$), (4) decreasing at S_2 ($case4$). It's obvious that $case2$ and $case3$ have no effect to the computation of application. So we only evaluate $case1$ and $case4$.

Results: Figure 3 shows the running time and the number of entries of push notification at the different number of flows. We can observe that our system supports incremental updates essentially. For $case1$, our system can skip the links with smaller c than the original value of the updated link.

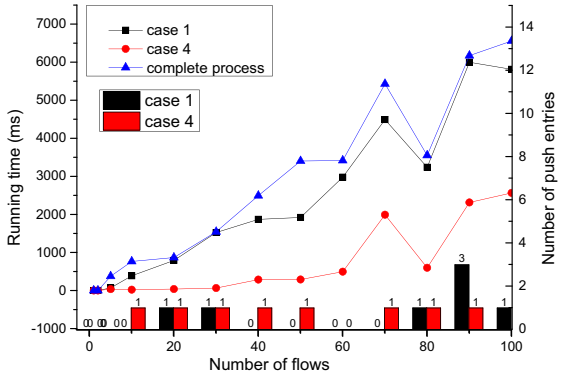

Figure 3: The running time and number of updates in push notification.

And for $case4$, our system should only consider S_1 and the updated one since the rest links in S_2 have no chance to add in S_1. It shows that the running time of updating of both cases is smaller than the complete one (line chart) and also the running time of $case1$ is higher than $case4$ since the problem set of $case4$ is S_1 plus the updated one which is much smaller than $case1$. Also the bar graph specifies the number of entry to push. 0 means that even the network OS receives updates from underlay network, the updates don't need to push to the application. Note that for $case2$ and $case3$, the number of entries for pushing should be all zero.

6. REFERENCES

[1] R. Alimi, R. Penno, and Y. R. Yang. *The ALTO Protocol, RFC 7285*, 2014.

[2] G. Bernstein, Y. Lee, W. Roome, M. Scharf, and Y. Yang. ALTO topology extension: Path vector as a cost mode. (IETF Internet-Draft), draft-yang-alto-path-vector-00.txt, 2015.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[4] Group-based Policy (GBP). `https://wiki.opendaylight.org/view/Group_Policy:Architecture/OVS_Overlay#Packet_Processing_Pipeline`.

[5] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[6] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[8] ONOS. `http://onosproject.org`.

[9] Open Networking Foundation. Openflow switch specification 1.4.0. Open Networking Foundation (on-line), Oct. 2013.

[10] OpenDaylight. `http://www.opendaylight.org`.

[11] S. Shenker. Software-defined networking at the crossroads. Stanford seminar, 2013.

[12] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98. ACM, 2013.