Shifter: A Consistent Multicast Routing Update Scheme in Software-Defined Networks

Guanhao Wu, Xiaofeng Gao[✉], Tao Chen, Hao Zhou, Linghe Kong, Guihai Chen
Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, 200240, China
{Tian_zwyxhi, tchen, h-zhou, linghe.kong}@sjtu.edu.cn, {gao-xf, gchen}@cs.sjtu.edu.cn

***Abstract*—Consistent routing update based on Software-Defined Networks (SDN) is a complicated problem due to the asynchronous and distributed data plane. Existing ordered update approaches mostly focus on the consistent routing update problem for unicast other than multicast, which should guarantee two consistencies, drop-freeness and duplicate-freeness. In this paper, we propose Shifter, a novel dynamic ordered update scheme for consistent multicast routing update based on SDN to guarantee both consistencies. Shifter advocates configuring inport match field in the forwarding rules to avoid duplicate. In order to guarantee drop-freeness, Shifter employs a dependency graph to dynamically schedule update operations, and uses a greedy solution to solve a subproblem named Replace Operation Tree Migration Problem (ROTMP). We conduct simulations to evaluate Shifter and find that Shifter can give a near optimal solution of ROTMP with very few rounds and little runtime for multicast routing update scenarios. To the best of our knowledge, Shifter is the first ordered update scheme to guarantee the two consistencies simultaneously.**

I. Introduction

Software-Defined Networking (SDN) is an advanced network architecture decoupling the control plane from the data plane, which allows the network managers to configure and update the network using the controller. SDN provides a global view of network and centralized computation for routing management. In spite of the logically centralized control, the switches still coexist in a distributed environment. During the routing update, the instructions from the controller to the switches take effect asynchronously, which may cause the loss of consistent properties such as loop, black hole, congestion and policy violation [1]. Nevertheless, as the logical centralization management of SDN, the controller can apply an update technique that carefully schedules the order of update operations to guarantee various consistencies.

Many consistent update techniques have been investigated in the literatures, such as *two-phase commit* [2]–[6] and *ordered update* [7]–[15]. *Two-phase commit (TPC)* approach adds a version tag to the rules and stamps packets with the new version tag in the header field. It is a powerful consistent update technique which guarantees the strong consistency like *per-packet consistency* [2]. However, it has to encode an irrelevant header field of packets with the routing version tag and costs double precious Ternary Content Addressable Memory (TCAM) memory [16]. *Ordered update* approaches find a sequence of operations updated in a well-designed order, which can be divided into static and dynamic approaches. The static approaches schedule all update operations as a sequence of operation sets executed round by round [9]–[13], while the dynamic ones maintain a dependency graph and execute dynamically-selected operations [7]. Besides, [17] combined *ordered update* and *two-phase commit* approaches to solve the policy-preserving update problem.

These proposed approaches solve consistent routing update problems with the awareness of various consistency, but most of them focus on unicast but less on multicast. Two-phase commit approach can guarantee the per-packet consistency of both unicast and multicast network update [2], but the header field occupied by TPC is not specially used for consistent updates. Therefore, this paper only focuses on ordered update approaches. The existing approaches on multicast routing update are not very effective. [18] proves that it is impossible to guarantee both drop-freeness and duplicate-freeness simultaneously by the previous ordered update approach.

The multicast routing update suffers transient **drop** and **duplicate** packets [18]. Dropped packets result in loss of data which could badly influence the Quality of Experience (QoE). Duplicate packets appear when replicated packets are forwarded into the same port, which consume double bandwidth. Besides, when the multicast route forms a loop, unlike the loop in unicast, a large amount of duplicate packets can be replicated rapidly, because some switches in the loop may replicate many copies of these packets and forward them back to the replicators. The useless packets will cause congestions on the ports of switches and receivers, and as a result, severe delay happens on the receivers and the application QoE is degraded. In addition, multicast protocols are based on UDP but not TCP, so the applications cannot rely on the underlying network protocols to eliminate the duplication. A good multicast application must maintain the numbers of recent received packets and check whether the incoming packets are duplicate ones, which may bring CPU overhead and delay jitter. Therefore, the network should try to void the duplication in case the applications cannot check duplication because of negligence of application designers.

In this paper, we propose **Shifter**, a novel ordered update scheme for consistent multicast routing update ensuring both **duplicate-freeness** and **drop-freeness**. In this paper, we write **in(e)gress port** as **in(out)port**. The reason why we solve the

978-1-5386-6043-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICNP.2018.00050

problem which is proved to be unsolvable [18] is that we study the necessary condition of the problem and employ a new rule model which makes it possible to satisfy the condition during the updates. The existing *ordered update* approach cannot guarantee duplicate-freeness without violating drop-freeness because the switch on both old and new routes cannot shift traffic from one inport to another by itself, because the rules in the switch do not have inport match field. This inspires us that the rules in switches should have the match field with inport, which is supported by Openflow protocol [19]. With this new rule model, the switch can shift traffic from one inport to another by itself using a *replace* operation, deleting an old rule and adding a new rule with same outport but different inports at the same time, without violating duplicate-freeness.

Based on the new rule model, it is easy to find all the update operations that need to be executed. The challenge is developing an algorithm to schedule the update operations with a minimum of time without violating drop-freeness. We observe that there are some dependency relations between different kinds of operations, which are *add*, *replace* and *delete* operations. We find two kinds of dependency, *Add-Replace* and *Replace-Delete*, which describe the orders between *replace* operations and *add* or *delete* operations. Then we can build a dependency graph with three layers to dynamically schedule three kinds of operations, so the execution time is no more than three rounds of the past static scheduling. In our simulation, over 80% of randomly generated scenarios can be solved by the dependency graph only considering the two dependencies.

However, for the rest scenarios, such a dependency graph is not enough and we address them by scheduling the order of Replace operations. We formulate this problem as Replace Operation Tree Migration Problem (ROTMP). The order of *replace* operations cannot be described by dependence relation, so we choose to statically schedule them round by round and integrate the solution into the dependency graph eventually. We propose a greedy solution which updates the maximum number of operations in each round and also formulate the ROTMP problem as a mixed integer program. We conduct simulations to compare the greedy solution with the optimal solution. The simulation results show that Shifter can give near optimal solutions for most update scenarios with average less than 1.2 rounds within 10 ms.

Specifically, our contributions are summarized as follows:

- We study the necessary condition to the solution and introduce the rule model with inport field to avoid duplicate-freeness. We also present a mechanism based on OpenFlow protocol to support the rule model.
- We study the dependency between different kinds of update operations and build a dependency graph to describe the dependency between different kinds of operations.
- We formulate the problem of scheduling *replace* operations as Replace Operation Tree Migration Problem and propose a greedy solution.
- We conduct simulations of Shifter and the results show that Shifter can give near optimal solutions for most update scenarios with very few rounds and little runtime.

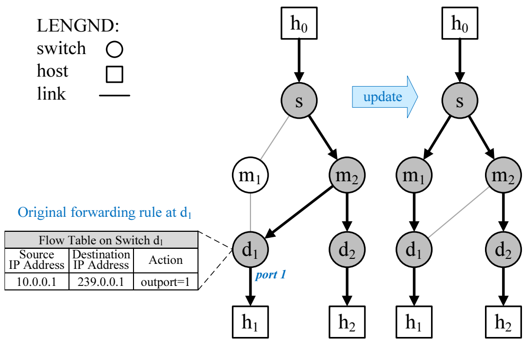
II. Preliminaries and objective

In this section, we introduce the drop-freeness and duplicate-freeness, and explain why the existing ordered update approaches cannot satisfy both of them. We indicate that the solution to the problem needs a necessary condition which can be supported by OpenFlow protocol [19], which inspires us to change the rule model. Then, we give the model of Consistent Multicast Routing Update Problem (CMRUP) based on new rule model. Finally, we present a method to generate appropriate different kinds of update operations.

A. Drop-freeness and Duplicate-freeness

Drop-freeness means the packets from the sender host can be forwarded to all receivers, and duplicate-freeness means the forwarding path from the sender to a receiver is unique. Take Fig. 1 as an example. The source host is h_0. Host h_1 and h_2 are the multicast group members. Each switch along the multicast trees has a flow table depicting the forwarding rule. Shown at d_1, once the forwarding rule matches the address of source host h_0 and the group address, it will forward the packets to port 1 in the *action* field.

Now we will update the route according to the initial and final multicast tree, and update the rules in switches s, m_1 and m_2. If m_2 is updated lastly, the update satisfies drop-freeness but duplicate will happen at d_1 because the packets will be replicated by s and be forwarded to d_1 via both m_1 and m_2. However, if m_2 is not updated lastly, the update satisfies duplicate-freeness but drop will happen when m_2 is updated before s and m_1. This example also illustrates that the two consistencies cannot be guaranteed simultaneously by ordered update approaches no matter what is the order.

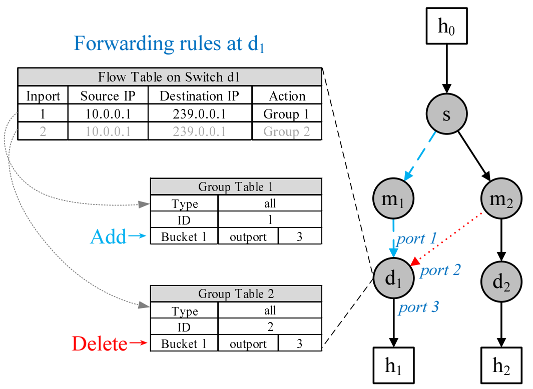


Flow Table on Switch d1		
Source IP Address	Destination IP Address	Action
10.0.0.1	239.0.0.1	outport=1

Fig. 1: A multicast routing update scenario

B. Necessary Condition

[18] claimed that the two consistencies cannot be guaranteed simultaneously by ordered update approaches, because (take Fig. 1 as an example again), the flow traversing through switch d_1 must change its inport transiently, but such a process has to be finished by at least two update operations on two switches (e.g. s and m_2). To guarantee both consistencies, we have to change the inport of a flow transiently by only one update operation on a switch, but not two different operations.



Flow Table on Switch d1			
Inport	Source IP	Destination IP	Action
1	10.0.0.1	239.0.0.1	Group 1
2	10.0.0.1	239.0.0.1	Group 2

Group Table 1		
Type	all	
ID	1	
Bucket 1	outport	3

Group Table 2		
Type	all	
ID	2	
Bucket 1	outport	3

Fig. 2: Flow table and group tables of Switch d_1. The dotted (dashed) lines represent the connection only in the old (new) route. The full lines are the connection in both routes.

Actually, we can satisfy such requirement by configuring the rules with the *inport* field and defining a *replace* operation to transiently change the inport of the flow. By this way, the switches can only accept the packets matching the *inport* field of rules. We temporarily call the update based on the rules with inport field *Inport-based update*. We can describe an inport-based consistent routing update process as follows:

In the update scenario of Fig. 1 when switch s and m_1 add their new rules, packets will come from s to d_1 via m_1, while d_1 will not accept these packets since it has no rule matching the inport connected with m_1. Then, d_1 can execute the Replace operation which deletes the old rule and adds the new rule. In addition, it will not forward duplicated packets to h_1. Finally, m_2 can delete the rule with outport to d_1 safely.

Only using flow table is not able to execute the above process because its *action* field of an entry can only be replaced integrally, but not be added or deleted partially. However, the key of *replace* operation is that we can add, delete or modify the rules taking **given inport and outport as a unit**. With the help of OpenFlow protocol [19], we can use flow tables and group tables to complete the configuration. An example is exhibited in Fig. 2 at switch d_1. We set the *action* field of each flow table entry matching an inport as a group table. Here two entries bring two group tables. The bucket in a group table can be individually added or deleted. In this scenario, the *replace* operation can be done by adding the bucket of outport 3 in Group 1 and deleting the the bucket of outport 3 in Group 2. With this new rule model, it is possible to guarantee both consistencies.

To compare the *inport-based update* with *oneshot update* which do all update operations together, we conduct a simulation using Mininet 2.0 [20] network simulation tool and Floodlight 1.2 [21] controller running on a PC with an Intel i5-7300hq quad-core processor. We develop two Floodlight modules to perform two kinds of updates in the update scenario of Fig. 1 – **oneshot** and **inport-based method**. Oneshot updates three switches s, m_1 and m_2 together while inport-based method updates s and m_1 in the first round, then d_1 in the second round and finally m_2 in the third round.

According to [7], the per-rule update time on a commodity switch is mostly less than 30 ms, but sometimes it can reach 100-200 ms. However, the simulation time it takes to update a rule in Mininet is always less than 10 ms. Therefore, we insert delay sentences before every update instruction in our programs and the length of the latency is randomly set between 100 ms and 200 ms.

During the simulation, host h_0 constantly sends UDP packets with a fixed rate to h_1 and h_2. We check the received packets in h_1 and find the amount of lost and duplicated packets. For each sending rate, we perform 100 times of updates. Fig. 3 shows the average number of lost and duplicated packets of two methods. As the sending rate increases, the lost and duplicated packets in oneshot increases proportionally. In contrast, the amounts in inport-based method keep less than 1. Therefore, inport-based method can surely reduce the packet loss and duplication. Note that there are a small amount of packets lost or duplicated in the inport-based method because the delay in the two paths from switch s to d_1 may be a little bit different, but the difference compared to the sum of controller reaction time, the controller-switch delay and rule installation time can be considered negligible.

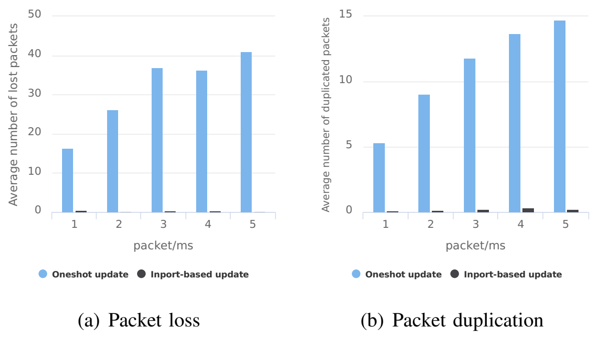


Fig. 3: Packet loss and duplication of oneshot and inport-based method in the update scenario of Fig. 1

C. Consistent Multicast Routing Update

The notations of basic elements are shown in Tab. I.

- **Inports and outports**: We use two nodes to represent a port in a switch. $ipt(v_i, s_j)$ means the inport on switch s_j which is oriented to a node v_i. $opt(s_j, v_k)$ means the outport on switch s_j which is oriented to a node v_k. $s_j.ipt/opts$ means the inport/outports in the rules on switch s_j.
- **Rules**: In inport-based update, an inport ipt and an outport opt decide a rule $r(ipt, opt)$. The essence of $r(ipt, opt)$ is that a flow table entry matching inport ipt and its action field points to a group table corresponding to ipt, which has a bucket whose action is $outport = opt$.
- **Update operations**: Update operations take given inport and outport as an abstract rule to modify the real rules in flow tables and group tables. Rule addition and rule deletion are trivial. Rule replacement is corresponding to two rules with the same outport but different inports.

There are some extended concepts as follows:

TABLE I: Notations of Multicast Routing Model

Notation	Description
h_0	Sender host
$h_i \in \{h_1, ..., h_n\}$	Receiver hosts
Is	Ingress switch
$s_j \in \{s_1, s_2, ..., s_m\}$	Switch
$v \in \{h_0, h_1, ..., h_n, Is, s_1, s_2, ..., s_m\}$	Node (including switches and hosts)
$ipt(s_i, s_j)$	Inport on switch s_j from s_i
$opt(s_j, s_k)$	Outport on switch s_j to s_k
$r(ipt, opt)$	Rule on s_j of inport ipt and outport opt
$add(r)$	Add operation of new rule r
$del(r')$	Delete operation of old rule r'
$rep(r, r')$	Replace operation of new rule r and old rule r'
$s_j.add/s_j.rep/s_j.del$	Add/Replace/Delete operation on switch s_j
$s_j.ipt/s_j.ipt'$	The new/old inport of switch s_j
$s_j.opts/s_j.opts'$	The new/old outports of switch s_j

- **Route**: The set of all nodes and the rules in the switches. It also represents a directed graph. If v_i and v_j are adjacent, v_i has a rule $r(., opt(v_i, v_j))$ and v_j has a rule $r(ipt(v_i, v_j), .)$, it has an edge (v_i, v_j).
- **Connection**: $v_i \rightarrow v_j$ means that the packets sent from v_i will arrive at v_j and v_j will not drop them. We say that v_j is connected to v_i in the route. Specifically, there exists a sequence $sq(v_i, v_j)$ of nodes and the rules in the nodes make the packets be forwarded from v_i to v_j via $sq(v_i, v_j)$.
- **Join switch**: the switch with different old and new inports.
- **Common receivers**: the receivers in both routes.

Now we can give the definition of **Consistent Multicast Routing Update Problem (CMRUP)**.

Definition 1 (CMRUP)**.** *Given two routes, which have the same h_0 and Is, generate the update operations and schedule their order to make the old route change into the new route, and the route during the update should guarantee that for any common receiver h_i, $h_0 \rightarrow h_i$ and there is only one $sq(h_0, h_i)$.*

Drop-freeness implies $h_0 \rightarrow h_i$, which is equal to so-called *relaxed-loop-freeness* [10] which means there are no loops in the path from source to destination. Duplicate-freeness implies there is only one $sq(h_0, h_i)$. CMRUP is more complex than the past consistent unicast routing update problem. The solution should generate all update operations first and then schedule them. The goal is to minimize the update time. In unicast update, only one operation needs to be executed in a switch, which directly changes the outport of the flow. However, in CMRUP, a switch has an old and a new inport, and several old and new outports. The two groups of outports may have intersection, so it is not evident to generate appropriate operations. Furthermore, in CMRUP, the effect of operations are more complex than the operations using the rule model that has not *inport* field. Besides, the receivers in CMRUP are multiple and they can be removed, added or maintained during the update, and only the common receivers need consistencies during the update.

III. Shifter design

In this section, we present the three main components of Shifter – generation of update operations, dependency graph of update operations, static scheduling of *replace* operations. First, we introduce how to generate proper update operations according to given two routes to avoid duplicate. Next, in order to avoid drop, we present the conditions of the dependency between *add* and *delete* operations with *replace* operations. We develop an algorithm to generate update operations and build the update operation dependency graph. The dependency graph could dynamically schedule the operations and solve a portion of scenarios. As for the rest scenarios, the key to the problem is to schedule *replace* operations.

However, the form of dependency graph cannot describe all feasible orders of *replace* operations, so we choose to statically schedule them. We formulate this problem as Replace Operation Tree Migration Problem (ROTMP) and propose a greedy solution for it. The solution of ROTMP is a sequence of the set of *replace* operations. Finally, we integrate the solution of ROTMP into the dependency graph by adding edges between *replace* operations in adjacent rounds. The execution of the final solution is dynamic but we can measure the solution by the number of layers in the solution. The dependency graph first has three layers of three kinds of operations. The operations in the same layers do not have dependencies with each other. However, the *replace* operations are divided into several sets according to the solution of ROTMP, so the actual number of layers is $1+x+1$, i.e., one layer of *add* operations, x layers of *replace* operations and one layer of *delete* operations.

A. Generation of Update Operations

A rule in the routes can be either added or deleted, if it does not exist in both routes. We can generate *add* or *delete* operations according to whether the rule is new or old. However, as Fig. 2 shows, some rules must be dealt with *replace* operations. We present the basic conditions that a new rule r and an old rule r' in switch s_j should be dealt with a *replace* operation as follows:

- Inports of r and r' are different and outports of r and r' is the same, according to the definition of *replace* operation.
- Assuming that the outport of r and r' is $opt(s_j, s_k)$, there exists at least one common receiver h_i, and $s_k \rightarrow h_i$ in both old and new routes.

Theorem 1. *If r and r' in a switch s_j satisfy the above two conditions, unless r and r' are dealt with $rep(r, r')$, drop-freeness and duplicate-freeness cannot be guaranteed simultaneously.*

Proof. First, we prove that if there exists at least one receiver host h_i and in both old and new routes $s_k \rightarrow h_i$, then $s_j \rightarrow s_k$ and it should be guaranteed that there is at most one $sq(h_0, s_k)$ during the update.

$h_0 \rightarrow h_i$ should be guaranteed during the update, so $sq(h_0, h_i)$ always exists during the update. Because in both old and new routes $s_k \rightarrow h_i$, if there are two $sq(h_0, s_k)$, there will be two $sq(h_0, h_i)$. Therefore, it should be guaranteed that there is only one $sq(h_0, s_k)$. In both routes, the next hop of s_j is s_k, so $sq(h_0, h_i)$ always includes s_j following with s_k. If $s_j \rightarrow s_k$ becomes false during the update, $sq(h_0, h_i)$ will be broken. Therefore, $s_j \rightarrow s_k$ should be guaranteed.

Second, we prove that if $s_j \to s_k$ and only one $sq(h_0, s_k)$ are guaranteed during the update, then r and r' should be dealt with $rep(r, r')$. If r is added by $add(r)$ and r' is deleted by $del(r')$ respectively, neither two orders of two operations can satisfy the requirements. Assume that inports of r and r' are $ipt(s_i, s_j)$ and $ipt'(s'_i, s_j)$. There are two cases:

- $add(r)$ happens first. If at the moment before $add(r)$ happens, $h_0 \to s_i$ and s_i is forwarding packets to s_j, there will be two $sq(h_0, s_k)$ after $add(r)$ happens. More detailedly, before $add(r)$ happens, $sq(h_0, s_k)$ via s'_i exists and when $add(r)$ happens, the old $sq(h_0, s_k)$ still exists because $add(r)$ cannot break it. Then, two $sq(h_0, s_k)$ will coexist. Otherwise, if at the moment before $add(r)$ happens $sq(h_0, s_k)$ via s_i does not exist, $sq(h_0, s_k)$ does not exist and thus $h_0 \to s_k$ cannot be guaranteed.
- $del(r')$ happens first so that $h_0 \to s_k$ cannot be guaranteed.

Therefore, $add(r)$ and $del(r')$ should happen at the same time, which must only be done by $rep(r, r')$. $rep(r, r')$ is the necessary condition for $h_0 \to s_k$ and only one $sq(h_0, s_k)$. □

If the generation of operations satisfies Thm. 1, the update guarantees duplicate-freeness because any two rules with different inports but identical outport could not coexist. The specific method for operation generation is integrated in Alg. 1.

B. Update Operation Dependency Graph

After generating all operations, it's time to schedule them with the aware of drop-freeness. As mentioned before, it is a complex problem for different kinds of operations and the multicast routing structures. We hope the order of operations can be described by a few constraints so that we can determine the order of every single operation simply. Fortunately, we have the following observation which implies how to schedule *add* and *delete* operations:

- $add(r(., opt(s_j, s_k)))$ and $del(r'(., opt(s_j, s_k)))$ could cause drop only when a join switch exists in the downstream of s_k in the old route.

In this paper, we consider the downstream of a switch s_j in the route, including s_j itself. When no join switches exist in the downstream of s_k in one route, there will be no common receivers, so the operations do not cause drop. Without the loss of generality, we only consider *add* operations. Assume that h_i is a common receiver and $s_j \to h_i$ via s_k in the new route. If the switches in $sq(s_j, h_i)$ of the new route are not join switches, $sq(h_0, h_i)$ of the old route must include $sq(s_j, h_i)$. However, $add(r(., opt(s_j, s_k)))$ should not be generated because r exists in both routes. Therefore, if no such join switches exist, and thus no common receivers exist, the operations will not cause drop.

We can demonstrate the observation in Fig. 2. d_1 is a join switch, so the *add* operations in s and m_1 and the *delete* operation in m_2 may cause drop if they are not carefully scheduled. The method to avoid violation of consistencies is to let the *add* operations happen before the *replace* operation in the join switch d_1 and the *delete* operations happen after $d_1.rep$. We describe it by an **Update Operation Dependency Graph (UODG)** – a dependency graph of update operations as nodes and the order between two operations as directed edges. With the UODG, the controller constantly checks and executes the operations in the UODG who have no in-coming edges. After getting a reply message from the switch, the controller deletes the corresponding operation and the edges connected to it in the UODG. The controller executes this process repeatedly until the graph becomes empty.

Now we give the conditions that an edge between two operations should be built in the UODG as follows:

(Add-Replace) $add(r(ipt(., s_j), opt(s_j, s_k)))$ must happen before $rep(r(ipt(s_a, s_b), opt(s_b, s_c)), r'(ipt'(s'_a, s_b), opt(s_b, s_c))$ is executed.

1) s_a is in the downstream of s_k in the new route.
2) There are common receivers in the downstream of s_c.
3) For any s_x followed with s_y in $sq(s_j, s_b)$, there is no edge from s_x to s_y in the old route.

(Replace-Delete) $del(r(ipt(., s'_j), opt(s'_j, s'_k)))$ must happen after $rep(r(ipt(s_a, s_b), opt(s_b, s_c)), r'(ipt'(s'_a, s_b), opt(s_b, s_c))$ is finished.

1) s'_a is in the downstream of s'_k in the old route.
2) There are common receivers in the downstream of s_c.
3) For any s_x followed with s_y in $sq(s'_j, s_b)$, there is no edge from s_x to s_y in the new route.

Before proof, we demonstrate the third condition by Fig. 4. There are 4 *add* operations and 4 *delete* operations in this scenario. We can find that there are no dependencies between the *add* operations in s and m_1 with the *replace* operation in d. In fact, d only depends on the *add* operations in m_4 and m_5. The reason is that $m_3 \to m_4$ in both routes, and therefore $h_0 \to m_4$ is guaranteed during the update. The connection in the upstream of m_4 is irrelevant with d. The *delete* operations are similar with the *add* operations.

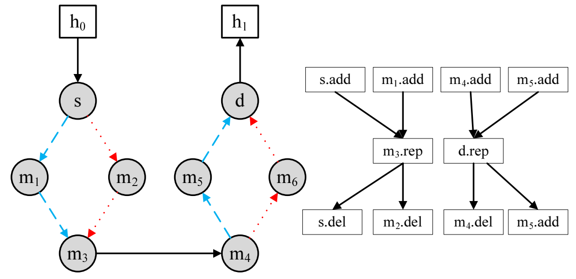


Fig. 4: An update scenario and its UODG. Operations in s, m_1, m_2 have no dependencies with the *replace* operation in d.

Now we explain the order between operations which satisfy the conditions. The relation between the two operations that the conditions describe can be summarized as s_j is connected with s_b by a path, whose edges only exist in either old or new routes. In addition, $h_0 \to s_k$ should be guaranteed during the update because of the second condition. Therefore, before *rep* happens, the old path cannot be broken by any *delete* operations in the switches in the old path, and the new path should be established by the *add* operations in the switches in the new path.

Note that the two kinds of dependency are necessary conditions for all feasible orders. The possible orders described by UODG must includes all feasible orders, and even in some scenarios they are identical. Furthermore, there are no more constraints on the orders of *add* and *delete* operations, because based on the current UODG they will not cause neither drop-freeness nor duplicate-freeness.

We combine the generation of update operations and the UODG into an algorithm which is a depth-first search algorithm which uses a recursive function called UODG-DFS (Alg. 1). The algorithm first starts from the ingress switch as the parameter of the function and finally return the update operation dependency graph.

Algorithm 1: UODG-DFS

```
   Input: $s_j$, $G$
1  if $s_j$ ∉ old route then
2      foreach $r(., opt(s_j, s_k))$ of $s_j$ in the new route do
3          G.add_node($add(r)$)
4          UODG-DFS($s_k$)
5  else
6      if $s_j.ipt == s_j.ipt'$ then
7          foreach $opt(s_j, s_k) \in s.opts'$ do
8              if $opt(s_j, s_k) \notin s.opts$ then
9                  G.add_node($del(r'(s_j.ipt, opt(s_j, s_k)))$)
10         foreach $opt(s_j, s_k) \in s.opts$ do
11             if $opt \in s.opts'$ then
12                 UODG-DFS($s_k$)
13             else
14                 G.add_node($add(r(s_j.ipt, opt(s_j, s_k)))$)
15                 UODG-DFS($s_k$)
16     else
17         foreach $r'(ipt'(s_i, s_j), opt(s_j, s_k)) \in s_j$ in the old route do
18             if $opt \notin s_j.opts'$ then
19                 G.add_node($del(r'(ipt'(s_i, s_j), opt(s_j, s_k)))$)
20         foreach $r(ipt(s_i, s_j), opt(s_j, s_k)) \in s_j$ in the new route do
21             if $opt \notin s_j.opts'$ then
22                 G.add_node($add(r(ipt, opt(s_j, s_k)))$)
23                 UODG-DFS($s_k$)
24             else
25                 G.add_node($rep(r(s_j.ipt, opt(s_j, s_k)),$
26                                 $r'(s_j.ipt', opt(s_j, s_k)))$)
27                 // Traverse along the new route upstream.
28                 $cur \leftarrow s_i$, $prev \leftarrow s_j$.
29                 while $cur$ is not the next hop of a join switch do
30                     G.add_edge($add(r(ipt(s_x, cur),$
31                                 $opt(cur, prev)))$,$rep$)
32                     $prev \leftarrow cur$, $cur \leftarrow s_x$.
33                 // Traverse along the old route upstream.
34                 // $s_j.ipt' = ipt(s'_i, s_j)$
35                 $cur \leftarrow s'_i$, $prev \leftarrow s_j$.
36                 while $cur$ is not the next hop of a join switch do
37                     G.add_edge($rep$,$del(r(ipt(s_x, cur),$
38                                 $opt(cur, prev)))$)
39                     $prev \leftarrow cur$, $cur \leftarrow s_x$.
40                 UODG-DFS($s_k$)
```

Alg. 1 omits the generation of some *delete* operations, which do not depend on *replace* operations. They can be easily added by traversing those switches only in the old routes. Alg. 1 checks the rules in a switch and generates *replace* operations according to Thm. 1 as well as *add* and *delete* operations. The pseudo-code from Line 27 and 39 traverse upwards along the new and old route to find the *add* and *delete* operations satisfy Add-Replace and Replace-Delete conditions, and then add directed edges between the operations. The time complexity of the DFS algorithm is $O(N+E)$, where N is the number of switches and E is the number of edges in both routes, because an edge can be traversed for three times at most.

We can make a summary of the generation of update operations and the UODG. First, the operations generated by Alg. 1 implies that the update will be duplicate-free, because every switch cannot keep two rules with the same outport but different inports, which must be dealt with a *replace* operation. Second, based on the UODG, *add* or *delete* operations have no other constraints, and only the order of the correlated *replace* operations involve consistencies, because $sq(h_0, h_i)$, where h_i is a common receiver, are only changed by *replace* operations. Finally, there are some scenarios that UODG cannot guarantee drop-freeness, which should be solved by static scheduling of the *replace* operations (e.g. Fig.5).

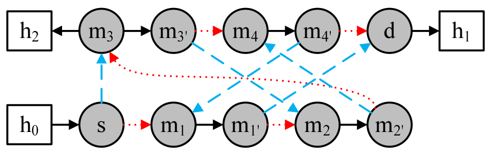


Fig. 5: An infeasible case for UODG

The *replace* operations in m_1, m_2, m_3, m_4 and d cannot be executed in an arbitrary order. It is obvious that $m_1.rep$ and $m_2.rep$ cannot be executed first, otherwise drop will happen. We carefully check all feasible orders of the 5 *replace* operations and find that no dependency relation can describe all feasible orders of them. Therefore, the *replace* operations should be statically scheduled in a sequence of subsets of them, updated round by round, based on the UODG.

C. Static Scheduling of Replace Operations

According to the UODG, every *replace* operation should be executed after a path from the ingress switch is finished by some *add* operations. At the moment before the *replace* operation happens, the switch actually receives packets from the new inport, though they are dropped because the new rule has not been added. However, after the *replace* operation happens, the path from h_0 to this switch may be broken. For example, in Fig. 5, after $m_{4'}.add$ finishes, $m_{4'}$ is sending packets to m_1 and then $m_1.rep$ can be executed according to the UODG. However, after $m_1.rep$ happens, m_1 cannot receive packets from s and $m_{4'}$ cannot either. A feasible order of the five *replace* operations can be $\{m_3, m_4, d\} \rightarrow \{m_2\} \rightarrow \{m_1\}$.

The reason for why we have to schedule the order of *replace* operations is that some *replace* operations can cause loops during the update. Such a loop must consist of a part of old route and a part of new route, because the old or new route cannot have any loops. The two parts of different routes are connected by two join switches (s_j and s_k). s_j is an upstream switch of s_k in the old route, and in contrast s_k is an upstream switch of s_j in the new route (e.g. m_2 and m_3 in Fig. 5).

This reason inspires us that the route during the update may be able to described as a tree which do not have loops between h_0 and common receivers. The unit of update is simplified as only one *replace* operation since that the orders of *Add* and *Delete* operations are correlated to it by the UODG. Besides, the switches that are not be updated by any *replace* operations can be ignored, except Is. Finally, the original problem is converted into a problem called *replace* Operation Tree Migration Problem (ROTMP), whose answer is the order of *replace* operations. First, we present the detailed definition of some related concepts about ROTMP as follows:

- **Replace Operation Tree (ROT)** is a directed graph including Is and the join switches. A ROT is transformed from a route. Two nodes, e.g. s_j and s_k, are connected from s_j to s_k in the ROT if and only if in the route $s_j \to s_k$ and $sq(s_j, s_k)$ has no other join switches except s_j and s_k (s_j can be Is). The ROTMP has two ROTs at the beginning, which are converted from the old and new routes, denoted as **OROT** and **NROT**. The nodes may have different father nodes in OROT and NROT. The migration process is to use *Migration operations* to migrate OROT to NROT.
- **Migration operation** is a combination of all *replace* operations in a switch. The ROT needs to be migrated from one to another by migration operations, as the route needs to be updated by update operations. In ROTMP, the *add* and *delete* operations are ignored because they are correlated to *replace* operations, and the corresponding *add* operations of a *replace* operations are integrated into a migration operation together with the *replace* operation. A migration operation in switch s_j will change the father node of s_j in the ROT.
- **Target Nodes** are the switches that should always be connected with Is in a ROT. In the route, common receiver(s) are in the downstream of a target node and the path from a target node and the common receiver has no other join switches. During the migration, all nodes have a father node and some intermediate ROTs may not only have a tree but also another separate part that does not have any target nodes so that common receivers are still connected to Is.
- **Concurrent Migration Operation Set (CMOS)** is a set of migration operations that can be executed in a round. When n migration operations in a CMOS are updated together, the ROT may be changed into one of $2^n - 1$ different ROTs. Every possible ROT should guarantee all target nodes connected with Is.

Now we give the definition of Replace Operation Tree Migration Problem (ROTMP) as follows:

Definition 2 (ROTMP)**.** *Given OROT and NROT converted from the old and new routes, find a sequence of CMOSs, which are disjoint subsets of all migration operations, to migrate OROT to NROT.*

ROTMP is similar to the static ordered update approaches, whose goal is to minimize the rounds of update, so we also regard the migration of a CMOS as a round of update though the final solution is dynamic. Note that to simplify the problem, a migration operation represents all *replace* operations in the switch other than one, which may slightly increase the number of rounds but the solution still exists.

Fig. 6 shows the ROTMP of the scenario in Fig. 5 and a solution to it. The shadowed nodes are the nodes which have not been migrated. The switches $m_{x'}$ where $x \in \{1, 2, 3, 4\}$ are removed because they do not have *replace* operations. The migration operation on m_1, for example, means that in CMRUP $m_1.rep$ happens after $m_4.add$ according to the UODG so that m_1 is connected to m_4 directly in the third round. It takes 3 rounds to migrate OROT to NROT. Note that the first CMOS has 3 operations and it can be proved that no matter what the order of these 3 operations is, the ROT can guarantee drop-freeness (h_1 and h_2 are connected to s).

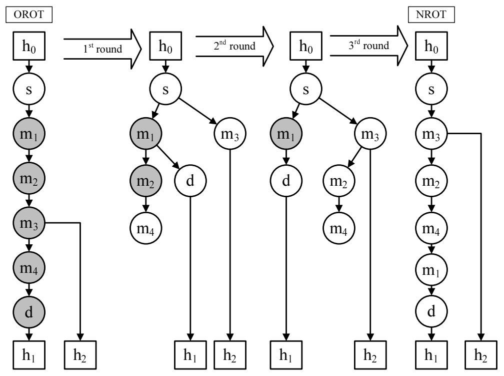


Fig. 6: ROTMP and its solution of the scenario in Fig. 5

Fig. 7 shows the whole solution of the scenario in Fig. 5. It connects the replace operations in the previous round to those in the next round in the UODG. This instance maybe seems to have too many layers, but the evaluation results show that such a complex case is very hard to appear.

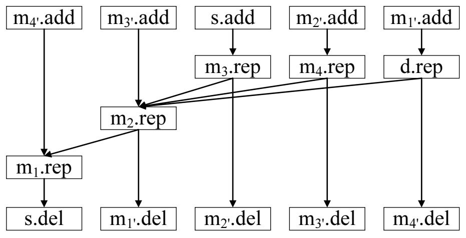


Fig. 7: Final solution of the scenario in Fig. 5.

In the next subsection, we formulate the maximum update problem and the optimal update problem as mixed integer programs.

IV. Greedy and optimal solutions

In this section, we present two mixed integer programs: *greedy program* to compute the maximum number of nodes to update in a round, and *optimal program* to compute the minimum number of update rounds. The greedy solution runs the greedy program in each round and updates the Replace Operation Tree (ROT) by its solution. We use randomly-generated

Replace Operation Tree Migration Problem (ROTMP) instances to compare their runtime and number of rounds.

A. Greedy Solution

During the update, the ROT is migrated to intermediate ROTs and the updated nodes could be removed to reduce the input scale of the mixed integer program in each round, while not changing the solution. We reconstruct the two ROTs by removing the updated nodes while keeping connectivity between every two remaining nodes changeless. Therefore, the input of the greedy program in each round is two ROTs which have the same nodes and every node except the source has different previous hop.

Now we formulate the maximum consistent multicast routing update problem as a mixed integer program. Given the two ROTs, we should choose the maximum Concurrent Migration Operation Set (CMOS). Every migration operation is corresponding with an old edge and a new edge which respectively connects with the old and the new father node. In fact, a set of migration operations can be a CMOS if and only if we can add the new edges into OROT without introducing any loop containing a node in an s-d path. The evidence is that every node except the source has only one father node. If there are no loops between the source and destinations, every target node can traverse upstream and finally reach the source. Otherwise, if a node in the s-d path is in a loop, the target node in that path will traverse along the loop and cannot reach the source. The reason why we can reserve the old edges is that the old and new incoming edges of a node cannot be in the same loop. From this inference, we can formulate the greedy program (1).

We denote the set of old edges in OROT as E_o and the set of new edges in NROT as E_n (let $E = E_o \cup E_n$), an the set of all updateable nodes as V. We also have a source s and a destination set D consisting of all target nodes.

$$\max \sum_{(u,v)\in E_n} x_{u,v} \tag{1}$$

$$\text{s.t.} \quad x_{u,v} = 1, (u,v) \in E_o \tag{1a}$$

$$z_{u,d} = x_{u,d}, d \in D, (u,d) \in E \tag{1b}$$

$$z_{u,v} \le x_{u,v}, (u,v) \in E \tag{1c}$$

$$z_{u,v} \ge z_{v,w} + x_{v,w} - 1, (u,v), (v,w) \in E \tag{1d}$$

$$|V| \cdot z_{u,v} + t_u - t_v \le |V| - 1, (u,v) \in E \tag{1e}$$

First, we use binary variables $x_{u,v} \in \{0,1\}$ to signify whether the edge $(u,v) \in E$ may appear during this round. The objective is to maximize the number of newly-added edges. Constraint 1a sets $x_{u,v} = 1$ when (u,v) is an old edge. Then, we set the binary variables $z_{u,v} \in \{0,1\}$ to indicate whether the edge (u,v) may appear and some target nodes are in its downstream. Constraint 1b sets $z_{u,d} = x_{u,d}$ as (u,d) contains a target node d. Constraint 1c enforce if $x_{u,v} = 0$, i.e., (u,v) does not appear, then $z_{u,v}$ should be 0. Constraint 1d embodies if $z_{v,w} = 1$ and (u,v) appears, then $z_{u,v}$ should be 1 because the downstream of (v,w) is also the downstream of (u,v). If $z_{v,w}$ or $x_{v,w}$ is 0, then the constraint is meaningless. Finally, $z_{u,v}$ represents the edges between the source and target nodes which should not form any loop. Therefore, constraint 1e sets the Miller-Tucker-Zemlin constraint [22] to avoid loops.

We can find out the variables in the objective whose value is 1 to decide the nodes to be updated in this round.

The greedy solution can be divided into three steps:

1) Run the program (1) with the input of OROT and NROT. Get the nodes to be updated.
2) Update the OROT with these nodes and reconstruct OROT and NROT by removing the updated nodes.
3) If OROT has no updateable nodes, then stop. Otherwise return to step (1).

B. Optimal Solution

We formulate the optimal update problem as a single mixed integer program to find the minimum number of rounds. First, we should assume that the number of rounds has an upper bound R_bound. Let $\Gamma = \{1, \cdots, R\_bound\}$. Then, we use $x^r_{u,v} \in \{0,1\},\ r \in \Gamma$ to represent whether the ROT contains (u,v) after the rth round. Specially, $x^0_{u,v}$ is corresponding with the OROT. Only when node v is updated in round r, $x^r_{u,v}$ is different from $x^{r-1}_{u,v}$. Therefore, constraint 2a enforces the lower bound of R. Constraint 2b and 2c implies every old edge will be deleted from the ROT once and only once. Constraint 2d relates the new edge to the old edge pointing at the same node. We use the binary variables $y^r_{u,v} \in \{0,1\}$ to signify whether (u,v) may appear before and after the rth round (see constraint 2e and 2f). Constraints 2e-2i are similar with constrains 1b-1e, which avoid loops between the source and destinations.

$$\min \quad R \tag{2}$$

$$\text{s.t.} \quad R \ge r \cdot (x^r_{u,v} - x^{r-1}_{u,v}), (u,v) \in E_n, r \in \Gamma \tag{2a}$$

$$x^{r-1}_{u,v} \ge x^r_{u,v}, (u,v) \in E_o \tag{2b}$$

$$x^{R\_bound}_{u,v} = 0, x^0_{u,v} = 1, (u,v) \in E_o \tag{2c}$$

$$x^r_{u,v} + x^r_{w,v} = 1, (u,v) \in E_o, (w,v) \in E_n \tag{2d}$$

$$y^r_{u,v} \ge x^r_{u,v}, (u,v) \in E \tag{2e}$$

$$y^r_{u,v} \ge x^{r-1}_{u,v}, (u,v) \in E \tag{2f}$$

$$z^r_{u,d} = y^r_{u,d}, d \in D, (u,d) \in E \tag{2g}$$

$$z^r_{u,v} \le y^r_{u,v}, (u,v) \in E \tag{2h}$$

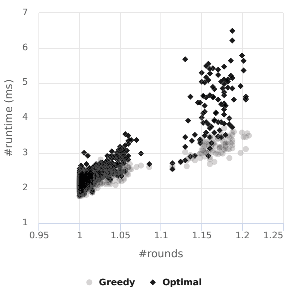
$$z^r_{u,v} \ge z^r_{v,w} + y^r_{v,w} - 1, (u,v), (v,w) \in E \tag{2i}$$

$$|V| \cdot z^r_{u,v} + t_u - t_v \le |V| - 1, (u,v) \in E \tag{2j}$$

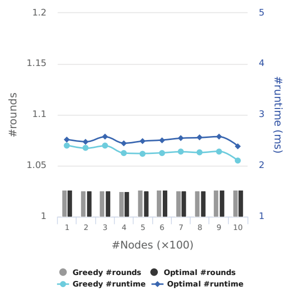
The program 2 does not guarantee feasible solutions unless R_bound is big enough. However, the program scale is proportional to R_bound and the runtime can increase exponentially with the growth of R_bound. Therefore, we tend to start from a small R_bound (e.g. 3) and try $R\_bound + 1$ next time.

C. Comparison

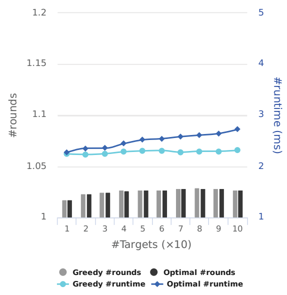
In consideration of the trade-off between runtime and round number, we should choose a proper solution according to the characteristics of the ROTMP instances. We explore the influence of three characteristics of a ROT on the two solutions,



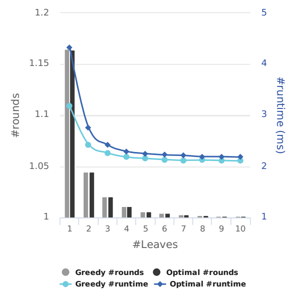
(a)



(b)



(c)



(d)

Fig. 8: Comparison between greedy and optimal solutions of ROTMP instances transformed from multicast routing update scenarios

including the total number of updateable nodes (written as **#Nodes**), the number of leaf nodes (written as **#Leaves**) and the number of target nodes (written as **#Targets**). We conduct numerical simulations with randomly generated ROTMP instances and run the two solutions with the same input. We use the state-of-the-art mathematical programming solver, Gurobi Optimizer 7.5 [23], to solve both mixed integer programs.

The ROTMP instances are randomly-generated according to the input parameters (#Nodes, #Leaves and #Targets). The generation can be divided into four steps:

1) Set s_0 as the source node of two ROTs. Generate a node set $\mathbf{S}$, $|\mathbf{S}| = \#Nodes$. Select a node set $\mathbf{D} \subseteq \mathbf{S}$ as the target nodes, $|\mathbf{D}| = \#Targets$.
2) In each ROT, randomly select a node s_i from $\mathbf{S}$ and randomly select a node s_j already in the ROT as s_i's father node, and guarantee that the number of leaf nodes in the ROT does not exceed #Leaves. Then remove s_i from $\mathbf{S}$. Repeatedly insert a node in $\mathbf{S}$ into the ROT until $\mathbf{S}$ is empty.
3) Remove the nodes which are not in any path from s_0 to a target node $s_i \in \mathbf{D}$.
4) Compare the two ROTs and recursively remove the nodes which have the same old and new father nodes and the non-common nodes that are not in both ROTs. When a node is removed, its father node should connect with its next hops.

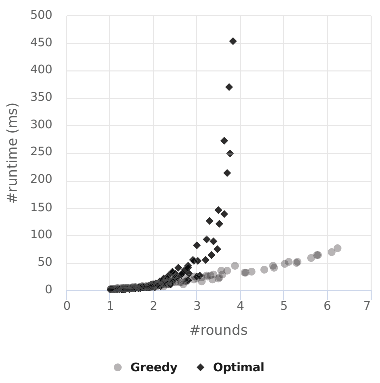


Fig. 9: Comparison between greedy and optimal solutions of directly generated ROTMP instances

The above process can guarantee randomness as much as possible. Note that these instances are not transformed from the consistent multicast update scenarios and the ROTMP instances transformed from the scenarios have much smaller problem scale and thus solutions when the original input number of nodes are the same.

We conduct 125 groups of experiments with different parameters. #Nodes, #Targets and #Leaves of these instances are respectively {10,20,30,40,50}, {1,2,3,4,5} and {1,2,3,4,5}. For each group of parameters, we generate 500 instances and test the average number of rounds (**#rounds**) and average runtime (**#runtime**) of greedy and optimal solutions.

Fig. 9 shows the interrelation between #rounds and #runtime of two solutions. When #rounds is small, #runtime of two solutions are close, but when #rounds becomes large, #runtime of optimal solution increases rapidly. Therefore, we can use the greedy solution instead of the optimal solution.

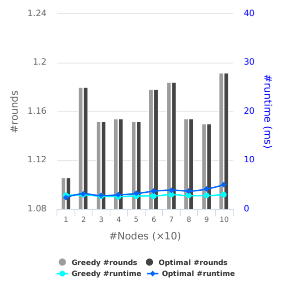
V. Evaluation

In this section, we perform simulations to compare the runtime and performance of greedy and optimal solutions in multicast routing update scenarios and explain that Shifter should use the greedy solution. We randomly generate multicast trees and update scenarios. The multicast trees can be generated by the same method of ROT. We use #Nodes, #Leaves and #Targets to describe the parameters of the multicast trees. For each group of parameters, we generate 500 instances of multicast routing update scenarios. Then we apply Alg. 1 to build the update operation dependency graph, and then test greedy and optimal solutions on the formulated Replace Operation Tree Migration Problem instances (ROTMP). Finally, we illustrates that the greedy solution can solve most update scenarios with very few rounds and little runtime.

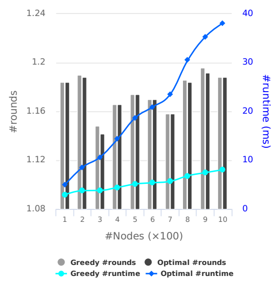
Fig. 8(a) shows the distribution of #rounds and #runtime of both solutions. Almost all points has less than 1.2 #rounds and 6 ms, so we can choose either greedy or optimal solution for this problem without much waste on runtime or update duration. Fig. 8(b) shows that #rounds and #runtime of both solutions do not have evident change with the growth of #Nodes. Therefore, it can be inferred that #Nodes may not have an evident influence on #rounds.

Fig. 8(c) shows that #rounds and #runtime increase slightly with the growth of #Targets. Since that the two paths from one source to a destination are likely to be different, there may exist a *replace* operation in the join switch of the two paths. Therefore, more destinations could increase the number of *replace* operations and thus the problem scale of ROTMP. Fig. 8(d) shows that #rounds and #runtime have a sharp decrease when #Leaves increases from 1 to 2, which explains the gap in the scatter points of Fig. 8(a). It implies that more complex routing structures make the nodes more disordered and decrease the probability that a node has the same previous hop and thus the problem scale.

Considering the impact of all parameters, when #Targets=#Nodes and #Leaves=1, the update scenarios could be the most difficult to update. We conduct simulations on small scale scenarios (see in Fig. 10(a)) and large scale scenarios (see in Fig. 10(b)) and let #Targets=#Nodes and #Leaves=1. We can see that in both small and large scenarios, #rounds of two solutions are very close and lower than 1.2. In small scenarios, #runtime of optimal solution is always less than 5 ms, but it increases much faster than greedy solution in large scenarios. Therefore, taking the runtime and the number of rounds into account, we should choose the greedy solution to solve the ROTMP. Fig. 10 also proves that the greedy solution can give very few rounds and little runtime for multicast routing update scenarios, even if the scenario is very large.



(a)



(b)

Fig. 10: Comparison between greedy and optimal solutions of ROTMP instances transformed from multicast routing update scenario instances (#Nodes=#Targets, #Leaves=1)

VI. Conclusion

In this paper, we propose Shifter, a novel ordered update scheme to address Consistent Multicast Routing Update problem based on Software-Defined Networks while ensuring both drop-freeness and duplicate-freeness. To the best of our knowledge, Shifter is the first ordered update scheme to guarantee drop-freeness and duplicate-freeness in consistent multicast routing updates. Shifter enables the inport field, which makes it possible to solve this problem, and employs an update operation scheduling algorithm that combines dynamic and static methods. We present a greedy solution to solve the static scheduling problem. The simulations prove that Shifter can perform near-optimal solutions for most update scenarios with very few rounds and little runtime.

VII. Acknowledgment

This work was supported in part by the Program of International S&T Cooperation (2016YFE0100300), the China 973 project (2014CB340303), the National Natural Science Foundation of China (Grant number 61472252, 61672353), the Shanghai Science and Technology Fund (Grant number 17510740200), and CCF-Tencent Open Research Fund (RAGR20170114). Xiaofeng Gao is the corresponding author.

References

[1] D. Li, S. Wang, K. Zhu, and S. Xia, “A survey of network update in SDN,” *Frontiers of Computer Science*, vol. 11, no. 1, pp. 4–12, 2017.
[2] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, “Abstractions for network update,” in *ACM SIGCOMM*, 2012, pp. 323–334.
[3] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven WAN,” in *ACM SIGCOMM*, 2013, pp. 15–26.
[4] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz, “zUpdate: Updating data center networks with zero loss,” in *ACM SIGCOMM*, 2013, pp. 411–422.
[5] N. P. Katta, J. Rexford, and D. Walker, “Incremental consistent updates,” in *ACM SIGCOMM HotSDN*, 2013, pp. 49–54.
[6] J. Zheng, H. Xu, G. Chen, and H. Dai, “Minimizing transient congestion during network update in data centers,” in *IEEE ICNP*, 2015, pp. 1–10.
[7] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, “Dynamic scheduling of network updates,” in *ACM SIGCOMM*, 2014, pp. 539–550.
[8] R. Mahajan and R. Wattenhofer, “On consistent updates in software defined networks,” in *ACM HotNets*, 2013, pp. 20:1–20:7.
[9] S. A. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid, “Transiently consistent SDN updates: Being greedy is hard,” in *SIROCCO*, 2016, pp. 391–406.
[10] A. Ludwig, J. Marcinkowski, and S. Schmid, “Scheduling loop-free network updates: It’s good to relax!” in *ACM PODC*, 2015, pp. 13–22.
[11] D. M. F. Mattos, O. C. M. B. Duarte, and G. Pujolle, “Reverse update: A consistent policy update scheme for software-defined networking,” *IEEE Communications Letters*, vol. 20, no. 5, pp. 886–889, 2016.
[12] K. Förster, R. Mahajan, and R. Wattenhofer, “Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes,” in *IFIP*, 2016, pp. 1–9.
[13] S. Dudycz, A. Ludwig, and S. Schmid, “Can’t touch this: Consistent network updates for multiple policies,” in *IEEE/IFIP DSN*, 2016, pp. 133–143.
[14] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, “Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies,” in *ACM HotNets, October 27-28*, 2014, pp. 15:1–15:7.
[15] W. Wang, W. He, J. Su, and Y. Chen, “Cupid: Congestion-free consistent data plane update in software defined networks,” in *IEEE INFOCOM*, 2016, pp. 1–9.
[16] A. X. Liu, C. R. Meiners, and E. Torng, “TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs,” *IEEE/ACM Transactions on Networking (TON)*, vol. 18, no. 2, pp. 490–500, 2010.
[17] S. Vissicchio and L. Cittadini, “FLIP the (flow) table: Fast lightweight policy-preserving SDN updates,” in *IEEE INFOCOM*, 2016, pp. 1–9.
[18] T. Kohler, F. Dürr, and K. Rothermel, “Consistent network management for software-defined networking based multicast,” *IEEE Transactions on Network and Service Management (TNSM)*, vol. 13, no. 3, pp. 447–461, 2016.
[19] “Openflow switch specification,” https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.
[20] “Mininet,” http://mininet.org/.
[21] “Floodlight,” https://floodlight.atlassian.net/wiki/.
[22] C. E. Miller, A. W. Tucker, and R. A. Zemlin, “Integer programming formulation of traveling salesman problems,” *J. ACM*, vol. 7, no. 4, pp. 326–329, Oct. 1960.
[23] “Gurobi,” http://www.gurobi.com/.